JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**Manuel Moser BSc**

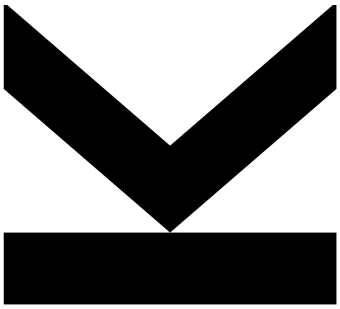
Submitted at
**Institute for Integrated Circuits:
Energy-Efficient Analog Circuits and Systems**

Supervisor
**Univ.-Prof. Dr. Harald Pretl**

Co-Supervisor
**DDI Patrick Fath**

May 2023

# Design of a Low-Power 12-bit Non-Binary Charge-Redistribution SAR-ADC utilizing the SKY130 Open-Source Technology



Master's Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Electronics and Information Technology

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

# Abstract

This work presents the design process of an integrated untrimmed 12-bit non-binary fully differential successive approximation register analog-to-digital converter (SAR-ADC) based on the SKY130 PDK using free and open-source software (FOSS).

The implemented 12-bit SAR-ADC is based on the design of a low-power 14-bit SAR-ADC for biosensing applications. The SAR-ADC consists of a segmented capacitive digital-to-analog converter (DAC), a self-clocking mechanism, an embedded common-mode voltage generation, a fully dynamic latch comparator, and a digital control block with an oversampling decimation filter to achieve a resolution of 16 bit using an oversampling factor of 256.

The DAC capacitor matrix is based on a 9-bit thermometer-coded and 3-bit binary-coded capacitor array while using a waffle-capacitor topology. The waffle-capacitor consists of 8 unit-capacitors per cell whereas one unit-capacitor contributes the capacity of $C^0 = 0.448\,\mathrm{fF}$, which results in a total capacitance of the DAC capacitor matrix of $C_{\mathrm{total}} = 1.83\,\mathrm{pF}$ per input.

Since the ADC is asynchronously self-clocked, it is saving energy while idling and no additional external clock is required. The clock-generation loop consists of configurable delay cells, which were realized with analog cells using the form factor of a SKY130 high-density digital standard cell. This allows parametrization of an analog circuit in a hardware description language (HDL) and hardening of the macro in an intentionally digital workflow. An additional power-saving switched capacitor voltage divider is used to generate the common mode voltage $V_{\mathrm{CM}}$ of the ADC.

A fully dynamic latch comparator is used to save energy since it only consumes power during a clock transition. To reduce the effect of comparator noise, an averaging filter for the least 4 significant SAR weights is included in the digital logic. An additional boxcar oversampling filter can be enabled and increases the conversion resolution from 12 physical bits up to 16 bits while using an oversampling rate of 256. The SAR code has been realized in a non-binary fashion for redundancy and error-correction capability.

With the proposed design choices, the ADC reached a sample rate of $f_s = 1.203\,\mathrm{MS/s}$ in the typical corner. Since the presented work is implemented as open-source, all sources are published on GitHub within the Apache 2.0 license.

In the first chapter, the workflow of open-source IC design using the SKY130 PDK will be introduced, followed by a SAR-ADC theory section where the functional principle of the differential SAR-ADC is torn down. The design hierarchy and performance goals are presented and explained in detail.

The second chapter covers analog designs, the generalized design flow, and analog designs in particular including the capacitive DAC core cell and other cells needed to assemble the DAC capacitor matrix, sample-and-hold switch, dynamic latched comparator, decoupling capacitors, and a switched-capacitor common-mode voltage generator. In addition, the analog macro implementations are described in detail including schematics and post-layout simulations.

In chapter 3, a method to design analog and time-dependent circuits using a digital workflow is proposed. The design and the implementation of time-dependent custom standard cells are presented. The circuit is described using hardware-description language (HDL) on the gate level where the custom standard cells can directly be instantiated. The digital workflow automatically generates the GDSII macro by placing and routing the analog standard cells into a standard cell grid.

Digital designs are the subject of chapter 4, where the introduction is performed with a description of the general design flow of RTL designs and the used simulation environment, followed by the digital top-level design and substantiation of the implemented hardware function blocks. Additionally, a method to simulate mixed-mode designs is presented.

Chapter 5 finalizes the design flow where all previously hardened macros and the digital logic are combined in a mixed-signal top-level design. The finalizing simulation is done with capacitive post-layout parasitic extraction using the parallel simulator Xyce, followed by the presentation of simulation results.

The last chapter, summary and outlook, recaps the obtained results from this work and discusses possible future work and limitations.

# Kurzfassung

Diese Arbeit behandelt den Entwurfsprozess eines integrierten, ungetrimmten, nichtbinären, volldifferenziellen, sukzessive Approximationsregister Analog-zu-Digital-Wandlers (SAR-ADC). Der SAR-ADC basiert auf dem SKY130 PDK unter Verwendung freier und quelloffener Software (FOSS).

Der implementierte 12-Bit SAR-ADC basiert auf einem low-power 14-Bit SAR-ADC Design für Biosensor-Anwendungen. Der Entwurf beinhaltet die Designs eines segmentierten, kapazitiven Digital-zu-Analog-Wandler (DAC), asynchronem Selbsttaktungsmechanismus, einer eingebetteten Referenzspannungserzeugung, einem volldynamischen Latch-Komparator und dem digitalen Logikblock mit Oversampling-Filter. Das Oversampling-Filter erlaubt die Erhöhung der ADC-Auflösung mit bis zu 256 Samples, dadurch kann eine Gesamtauflösung von bis zu 16 bit erreicht werden.

Die DAC-Kondensatormatrix basiert auf 9 Bit thermometer-code und 3 Bit binärkodierten Kondensatorarrays, die aktiven Kondensatoren verwenden eine Waffel-Topologie. Der Waffelkondensator besteht aus 8 Einheitskondensatoren pro Zelle, wobei ein Einheitskondensator eine Kapazität von $C^0 = 0.448\,\mathrm{fF}$ besitzt. Die Gesamtkapazität der DAC-Kondensatormatrix ist $C_{\mathrm{total}} = 1.83\,\mathrm{pF}$ pro Komparator-Eingang.

Da der ADC asynchron-selbstgetaktet ist, spart er im Leerlauf Energie, weil kein zusätzlicher Takt von außen eingespeist werden muss. Die Takterzeugungsschleife besteht aus konfigurierbaren Signalverzögerungszellen, die unter Verwendung des Formfaktors einer digitalen high-density SKY130-Standardzelle realisiert wurden. Dies ermöglicht die Parametrisierung einer zeitabhängigen Schaltung in einer Hardware-Beschreibungssprache (HDL) und die Generierung des Makros in einem digitalen Workflow. Ein stromsparender switched-capacitor Spannungsteiler wird zur Erzeugung der Referenzspannung $V_{\mathrm{CM}}$ verwendet.

Ein volldynamischer Latch-Komparator wird verwendet, da er nur während eines Taktübergangs Strom verbraucht und daher den Leistungsverbrauch niedrig hält. Um die Auswirkungen des Komparatorrauschens zu verringern, ist in der digitalen Logik ein Mittelwertfilter für die niederwertigsten 4 SAR-Gewichte enthalten. Ein zusätzliches Boxcar-Oversampling-Filter kann aktiviert werden um die Wandlerauflösung von 12 physikalischen Bits auf 16 Bit bei einer Oversampling-Rate von 256 zu erhöhen. Der SAR-Code wurde nicht-binär realisiert, um Redundanz und Fehlerkorrektur zu ermöglichen.

Mit den vorgeschlagenen Design-Entscheidungen erreicht der ADC unter typischen Bedingungen eine theoretische Abtastrate von $f_s = 1.203\,\mathrm{MS/s}$. Da die vorgestellte Arbeit quelloffen implementiert ist, werden alle Daten auf GitHub unter der Apache 2.0-Lizenz veröffentlicht.

Im ersten Kapitel "Introduction" wird der Workflow des Open-Source-IC-Designs mit dem SKY130 PDK vorgestellt, gefolgt von einem SAR-ADC-Theorieteil, in dem das Funktionsprinzip des differentiellen SAR-ADCs aufgeschlüsselt wird, und einem hierarchischen Designplan, der ebenfalls detailliert erläutert wird.

Das zweite Kapitel "Analog Design" behandelt analoge Designs, den verallgemeinerten Entwurfsablauf und analoge Designs im Detail. Die analogen Designs sind die DAC-Zellen, der Sample-and-Hold-Schalter, der volldynamische Latched-Komparator, die Entkopplungskondensatoren und der switched-capacitor Referenzspannungsgenerator. Darüber hinaus werden die analogen Makro-Implementierungen im Detail beschrieben, einschließlich Schaltpläne und Simulationen.

In Kapitel 3 "Digital-Friendly Analog Design" wird ein spezieller Design-Workflow für die Einbettung benutzerdefinierter, analoger Standardzellen vorgestellt. Diese angepasste Standardzelle nutzt den Formfaktor einer digitalen Standardzelle. Dies ermöglicht die automatisierte Generierung einer parametrisierten, asynchronen und analogen Makrozelle. Dafür wird eine Hardware-Beschreibungssprache (HDL) auf Gatterebene verwendet, unter Ausnutzung der Toolchain eines digitalen Workflows.

"Digital Design" ist das Thema von Kapitel 4. Die Einführung in das Kapitel beginnt mit einer Beschreibung des allgemeinen Entwurfsablaufs von RTL-Designs und der verwendeten Simulationsumgebung. Danach folgt die Spezifizierung vom digitalen Top-Level-Design und die Beschreibungen der implementierten Hardware-Funktionsblöcke. Zusätzlich wird eine Methode für Mixed-Mode Simulationen vorgestellt.

Kapitel 5 schließt den Entwurf des SAR-ADC "Top Level" ab, indem alle zuvor generierten Makrozellen und die digitale Logik in einem Mixed-Signal-Top-Level-Design kombiniert werden. Die abschließende Simulation erfolgt mit kapazitiver, parasitärer Post-Layout-Extraktion unter Verwendung des Parallelsimulators Xyce. Anschließend folgt die Präsentation der Simulationsergebnisse.

Das letzte Kapitel, "Summary and Outlook", fasst die in dieser Arbeit erzielten Ergebnisse zusammen und erörtert mögliche zukünftige Arbeiten und aktuelle Einschränkungen.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acknowledgement

If there was a general recipe for a Master's thesis, it surely would include hard work, enthusiasm, and maybe an extra cup of coffee for long night shifts. But, the most important ingredient of this recipe would be the people who have positively influenced me and this Master's thesis. It is time to express my gratitude for these people, especially to the people mentioned below.

First and foremost, I would like to thank my supervisor Univ.Prof. Dr. Harald Pretl for the introduction and the opportunity to work on this topic. For reviewing the progress, his help when I needed his guidance and the fruitful discussions that shaped the current design implementations and features.

I also want to express my special thanks to co-supervisor DDI Patrick Fath for his assistance and our enthusiastic exchanges regarding the theory of this work, for his guidance to acquire a scientific writing style, and also for proofreading this work multiple times.

The whole Open-Source ecosystem also deserves to be mentioned for its movement towards democratizing chip design and making it accessible for everyone. Without the Open-Source enthusiasts who help newbies and others, by answering questions, resolving issues, and their support regarding issues and feature requests, the topic of this thesis would not have been possible.

Many thanks to my colleagues from the Institute for Integrated Circuits (IIC-JKU), my fellow students, and my friends, who have made the last years much more enjoyable.

Finally, but most importantly, none of this could have happened without my family. Especially to my mother, for her unconditional support throughout my whole life, and for making my educational path possible in the first place: Thank you.

# Chapter 1

# Introduction

150 projects have been submitted to the Efabless MPW-8 (Multi Project Wafer) shuttle program [1] at the end of 2022, where participants get the chance of receiving free silicon to promote open-source chip design. Back in 2020, there were only 37 submissions in MPW-1 [2]. This trend indicates rising interest in open-source IC design amongst universities and enthusiasts.

The SAR-ADC design described in this work is based on the open-source technology SKY130. It has also been successfully submitted for tapeout in MPW-8 [3], whereas the sources are published on GitHub [1] within the Apache 2.0 license.

This chapter includes a brief introduction to the SKY130 PDK to describe the used process and libraries. The introduction is continued with a generalized theory of the SAR-ADC. The last section summarizes the specified design goals for this work.

## 1.1 SKY130 PDK

SKY130 is an open-source hybrid 180 nm-130 nm process design kit (PDK) that results from a collaboration between Google and SkyWater Technology Foundry [4]. The most important facts for this work are summarized in this section.

The core voltage is 1.8 V, and higher voltages are supported but in this work, they are not in use. For interconnections there is 1 local interconnect layer (TiN) below 5 metal layers (Al) available, solder bumps are connected over the redistribution layer (Cu) above `metal5`. Capacitors can be realized on 2 dedicated MiM (Metal-Insulator-Metal) layers. The complete layer stack diagram is shown in Fig. 1.1.

Regarding MOSFET device sizes one should know that SKY130 is a 180 nm-130 nm process, but the minimum-length device in the SKY130 process is 150 nm [5]. In addition, the 1.8 V-MOSFET device widths have special limitations, like the DRC rule that the transistor gate width must be $W \geq 0.42\,\mu$m, except in digital standard cells where the width can be reduced to $W \geq 0.36\,\mu$m, and $W \geq 0.15\,\mu$m in SRAM

[1] `https://github.com/iic-jku/SKY130_SAR-ADC1`

macros [6]. However, in this work DRC-compliant devices with $W \geq 0.42\,\mu$m and length $L \geq 0.15\,\mu$m were used for the analog layouts.

This SAR-ADC design uses the digital high-density library `_hd_` and the analog primitive cell library `pr`. The SKY130 PDK libraries follow a predefined name format for identification [7] specifying their process, source, type and name by abbreviations. For example, the name of the high-density library for digital standard cells `sky130_fd_sc_hd` resolves to the following properties:

- Process name: SKY130
- Library source abbreviation: `fd` = the SkyWater foundry
- Library type abbreviation: `sc` = standard cell
- Library name: `hd` = high density

This library includes the technology files for the development, simulation, and generation of high-density digital circuits. Digital designers can choose from various digital SKY130 libraries and use them to their needs, the SkyWater SKY130-PDK provides the following options:

- `sky130_fd_sc_lp`: low power
- `sky130_fd_sc_ls`: low speed
- `sky130_fd_sc_ms`: medium speed
- `sky130_fd_sc_hs`: high speed
- `sky130_fd_sc_hd`: high density
- `sky130_fd_sc_hdll`: high density with low leakage
- `sky130_fd_sc_hvl`: high voltage

For analog circuits the library `sky130_fd_pr` is available, it contains primitive models and cells (e.g., parametrizable MOSFETs, diodes, resistors, capacitors, fixed size cells) for the design and simulation of analog circuits. The included SPICE models allow PVT-variation simulations in slow, typical and fast corners, temperature variation and Monte Carlo simulation.

## 1.2 SAR-ADC Theory

The Successive Approximation Register ADC (SAR-ADC) [8] relies on the principle of a weighting procedure, whereby the analog value is step-by-step approximated by testing a sequence of known weights. The block diagram in Fig. 1.2 shows the basic structure of a SAR-ADC. In sampling mode, the sample-and-hold (S & H) switch is closed and $V_p$ follows $V_{\text{in}}$. In hold mode the switch is open and the capacitor keeps $V_p$ constant. The digital logic block sets the DAC code (weights) for the first comparison. As a result, this code is converted to an analog voltage at $V_n$, ideally proportional

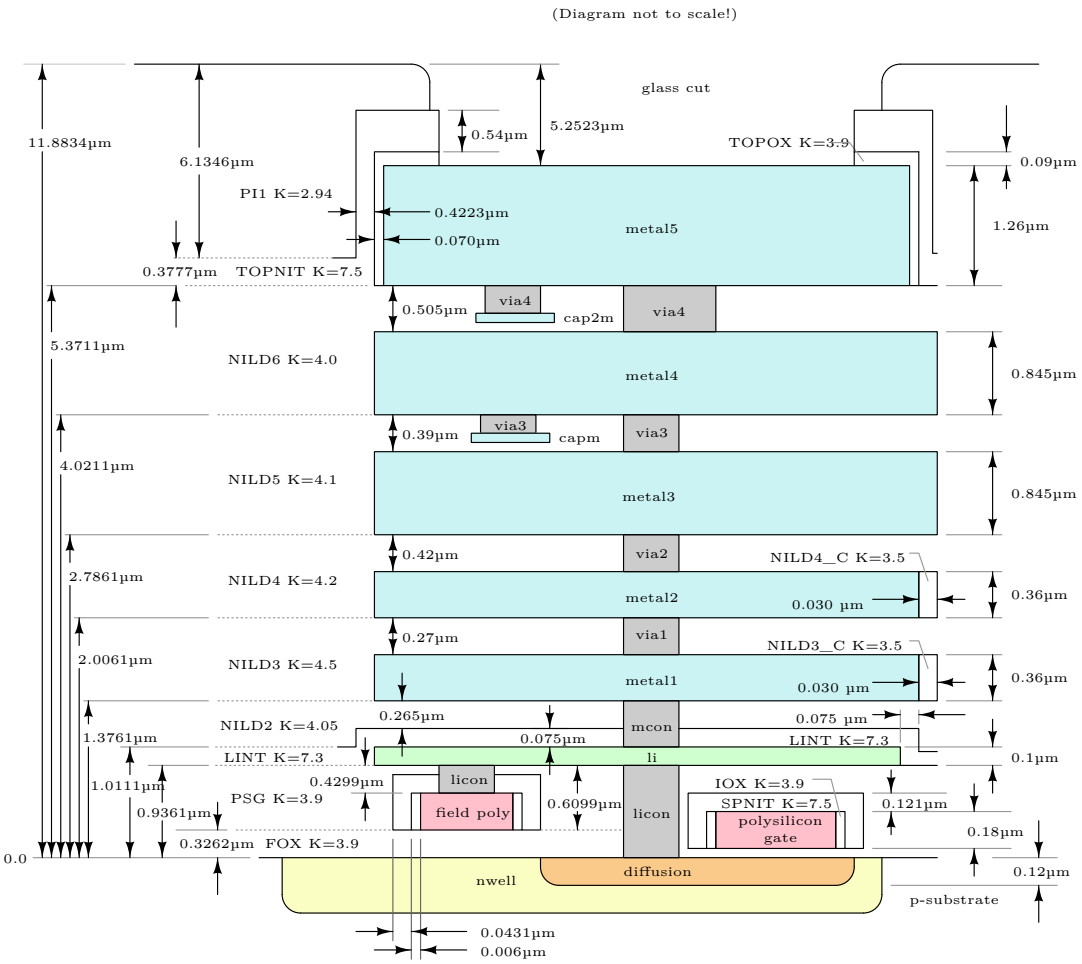



**Figure 1.1:** Layer stack diagram of the SKY130 PDK. Source: `https://github.com/google/skywater-pdk/blob/main/docs/_static/metal_stack.svg`

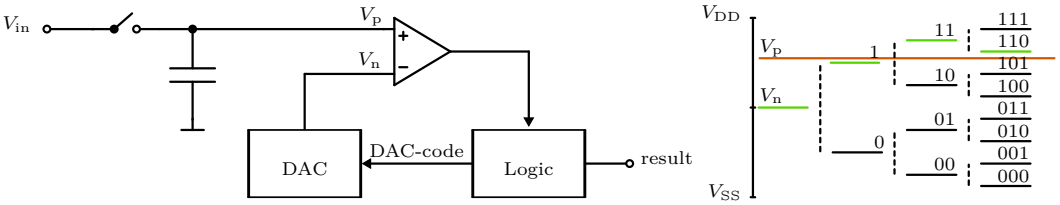
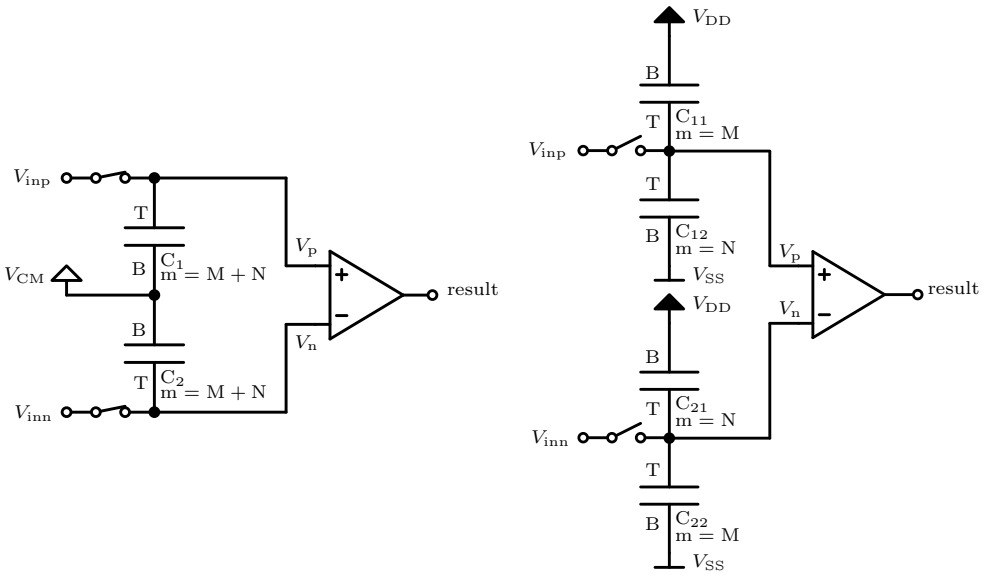



**Figure 1.2:** Generalized structure of the SAR-ADC. The S & H switch is closed in sampling mode and open in hold mode. In hold mode, $V_p$ is held constant by the capacitor. The logic block sets a DAC code on basis of the comparator result, the DAC converts the DAC code to an analog voltage and the comparator reports which input has the higher potential. On the right is an example of the possible decision paths for a 3-bit conversion with result `110`.

to its digital value. The comparator reports if the sampled voltage $V_p$ is greater or less than $V_n$, whereas digital logic evaluates the comparison result and calculates suitable weights for the next comparison. This procedure is repeated with progressively decreasing weight sizes until the combination of weights for the best approximation of $V_p$ has been evaluated. The graph in Fig. 1.2 shows a possible decision path for a 3-bit SAR-ADC with 3 comparisons and the result `110`.




**(a)** sampling-mode **(b)** conversion mode

**Figure 1.3:** Functional principle of the differential S & H and DAC-function. (a) ADC is in sampling mode. The capacitor bottom plates `B` are tied to common-mode voltage $V_{CM}$, and the top plates `T` to either $V_{inp}$ or $V_{inn}$. (b) The ADC is in conversion mode. The S & H switches are used to isolate the capacitor top plates from $V_{inp}$ and $V_{inn}$, and the only connection to the top plates is at the high-impedance comparator inputs.

The principle of conversion does not change in a differential application, however, the functional blocks must be mirrored on both comparator inputs which means both signals $V_{inp}$ and $V_{inn}$ need S & H switches with capacitors, and differential DAC voltages must be added on top of both sampled voltages. Fig. 1.3 shows how both functions (S & H and DAC) are realized in the differential SAR-ADC using a capacitive DAC with common capacitor top plates [9]. In sample phase (a) the capacitor top plates follow the input voltages $V_{inp}$ and $V_{inn}$ while the bottom plates are set to the common-mode voltage $V_{cm}$. In the following hold phase (b) the S & H switches are opened, which stores the voltage on the capacitors and isolates the top plates from $V_{inp}$ and $V_{inn}$. As a result, the only connection of the top plates is to the high-impedance input of the comparator.

If the voltage is changed at the bottom plates of the capacitors, then this voltage change is reflected on the top plates. Using figuratively $M + N$ capacitors of the same size with independent bottom plate switches to either $V_{DD}$ or $V_{SS}$ forms a capacitive voltage divider, the middle-node voltage of this voltage divider (capacitor top plates) can be lowered or increased in $M + N$ linear steps which correspond to the function of a digital-to-analog converter. The differential voltage $V_{inp} - V_{inn}$ can thus be manipulated in the voltage range $\pm(V_{DD} - V_{SS})$. This allows the realization of a SAR-ADC by setting a series of weights $M$ and $N$, which forces voltage change on the previously sampled input voltage by charge-redistribution, followed by evaluating the comparator results. Simultaneously this also means the capacitors act as a hold capacitor and DAC capacitor at the same time, which conveniently saves power and area [10].

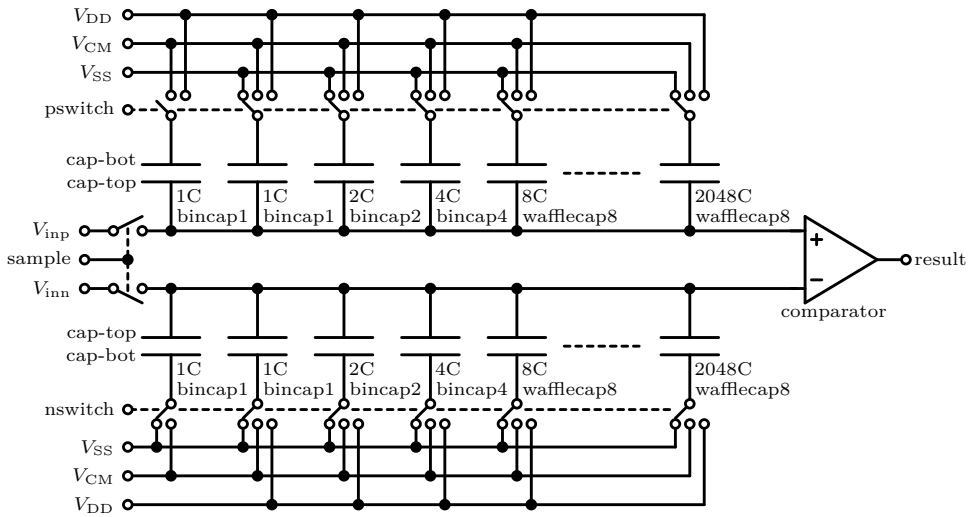



**Figure 1.4:** Working principle of the differential SAR-ADC with binary capacitor batches. In sample mode, the top plates are connected to $V_{\text{inp}}$ and $V_{\text{inn}}$, and the bottom plates to $V_{\text{CM}}$. In hold mode, $V_{\text{inp}}$ and $V_{\text{inn}}$ are isolated from the top plates, the sample switches are open, and the bottom plates are connected to either $V_{\text{DD}}$ or $V_{\text{SS}}$.

Fig. 1.4 further details the functional structure for a 12-bit SAR-ADC with binary weights 1/2, 1/4, ..., 1/4096. The capacitor batches reflect the binary SAR code weights, the first batch $2048 \times C$ represents 1/2 of the total weight which maps to the MSB `0x800`. The second batch $1024 \times C$ equals 1/4 weight and maps to `0x400`. This sequence is repeated until the least significant bit (LSB) capacitor $1 \times C$ tests the LSB bit `0x001`. After 12 cycles the differential input voltage has been approximated and the next sample & hold cycle begins.

In a binary SAR-ADC, the DAC weights use a factor of 2 (`1, 2, 4, 8, ...`), but a charge redistribution ADC can also be built with smaller and non-binary factors (e.g., 1, $1.85^1$, $1.85^2$, $1.85^3$, ...). The non-binary algorithm is described in detail in Sec. 4.3.2. Subsequently, the number of clock cycles needed for the binary algorithm is exactly the ADC resolution. In contrast, the non-binary SAR-ADC needs additional clock cycles, but this drawback can be compensated by increasing the frequency [10]. The decreased weight factor introduces some error tolerance, small errors do not affect the result if there are multiple search paths for the same input signal available.

The algorithm for a single-ended SAR-ADC has been described in [12], the given formulas were adapted to be valid for a fully differential SAR-ADC. If $M \in \mathbb{N}$ is the resolution of the fully differential SAR-ADC, $\Theta[k] \in \mathbb{N}$ is the $k^{th}$ weight of $N$ weights while $N$ is limited to $\{N \in \mathbb{N}_0 : N \geq M\}$, and $\mathrm{s}[k]$ is the $k^{th}$ comparison result with the following conditional value:

$$\mathrm{s}[k] = \begin{cases} 1, & \text{if } V_{\text{inp}} - V_{\text{inn}} > 2 \cdot V_{\text{ref}}[k] \\ -1, & \text{otherwise} \end{cases} \tag{1.1}$$

The number of active capacitive DAC cells $d_{\text{DAC}}[k]$ determines the analog reference voltage $V_{\text{ref}}[k]$. In a differential SAR-ADC there are 2 DACs connected to both

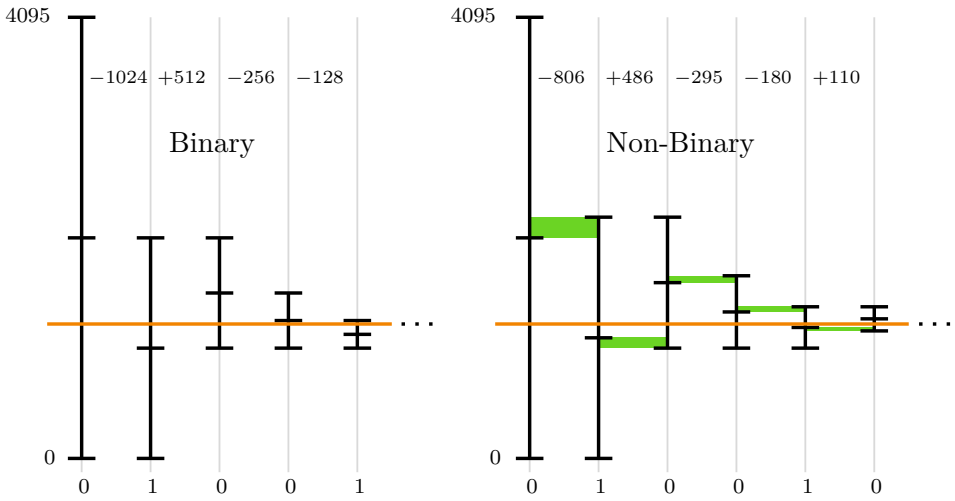



**Figure 1.5:** Difference between 12-bit binary and non-binary SAR-algorithm, visualized are the search-intervals per conversion step and the resulting redundant areas in the non-binary case. Figure obtained from [11].

comparator inputs. The sign of the weights in each DAC is determined by the comparator input. The value for $d_{\mathrm{DAC}}[k]$ at the $k^{th}$ step is given by:

$$d_{\mathrm{DAC,inn}}[k] = 2^{M-1} + \sum_{i=2}^{k} \mathrm{s}[i-1] \cdot \Theta[i] \tag{1.2}$$

$$d_{\mathrm{DAC,inp}}[k] = 2^{M-1} - \sum_{i=2}^{k} \mathrm{s}[i-1] \cdot \Theta[i] \tag{1.3}$$

The result $d$ for the non-binary $M$-bit SAR-ADC using $N$ weights is in the set $\{d \in \mathbb{N}_0 : 0 \leq d \leq 2^M - 1\}$ and is calculated using the generalized formula:

$$d = 2^{M-1} + \sum_{i=2}^{N} \mathrm{s}[i-1] \cdot \Theta[i] + \frac{1}{2}(\mathrm{s}[N] - 1) \tag{1.4}$$

The sum of all weights $\Theta_k$ must equal the maximum binary value of the ADC to ensure that the full range of the ADC is reachable [11].

$$\sum_{k=1}^{N} \Theta_k = 2^M - 1 \tag{1.5}$$

## 1.3 Design Goals

The design goals in this work are based on a previously designed 14-bit SAR-ADC [9] with the block diagram shown in Fig. 1.7. The proposed SAR-ADC uses a capacitive DAC matrix in a 180 nm process. Key features of this ADC are: Programmable delays for an asynchronous and self-clocked loop, averaging of the LSB bits, oversampling, thermometer-code and binary capacitive DAC cells, and a non-binary SAR algorithm.

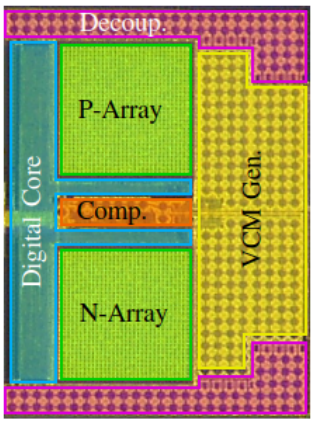



**Figure 1.6:** The layout of the referenced 14-bit SAR-ADC [9] used as a template for the floorplan.

A top-down-bottom-up design flow was used for this design. The first step is to partition the top-level hierarchy and the floorplan according to the specifications (top-down), which divides the top level into a hierarchical structure of smaller design elements, while the floorplan gets roughly defined. This is already done for most parts by using the referenced SAR-ADC [9] as a design template.

The flow is continued in reverse order (bottom-up) by designing the analog macrocells at the lowest hierarchical level. In this work, the first implemented analog cell was the DAC capacitor structure and its decoder circuit in the DAC core cell. As the layout of the DAC cell is finished, the other cells (dummy, drive, binary, gate) are designed together with the decoupling capacitor `adc_noise_decoup_cell.mag`, and as a result, the capacitive DAC matrix can be assembled (`adc_array_matrix_12bit.gds`). Then, the symmetric NOR gate, the comparator, the latch, and the combined latched-comparator `adc_comparator_latch.gds` have been designed. The decoupling capacitor is also used for the $V_{CM}$ generator `adc_vcm_generator.gds`. For the clock generator, a semi-analog delay cell using a digital standard cell form factor and a digital RTL2GDSII workflow is used to harden the synthesizable clock-generator layout with programmable delays `adc_clkgen_with_edgedetect.gds`. Finally, the digital RTL is designed using a hardware description language and all the previously designed analog macros can be combined with the digital logic in the top-level design `adc_top.gds` using the RTL2GDSII workflow.

Following is a summarized list of the design goals:

- Usage of the SKY130 minimum length MOSFETs whenever possible (150 nm) to minimize area and power consumption.
- Fully differential design to minimize common-mode influence.
- Synthesizable, asynchronous and self-clocked operation with an embedded clock generator `adc_clkgen_with_edgedetect.gds` with programmable delays to reduce power consumption and allow configuration of the sample rate.

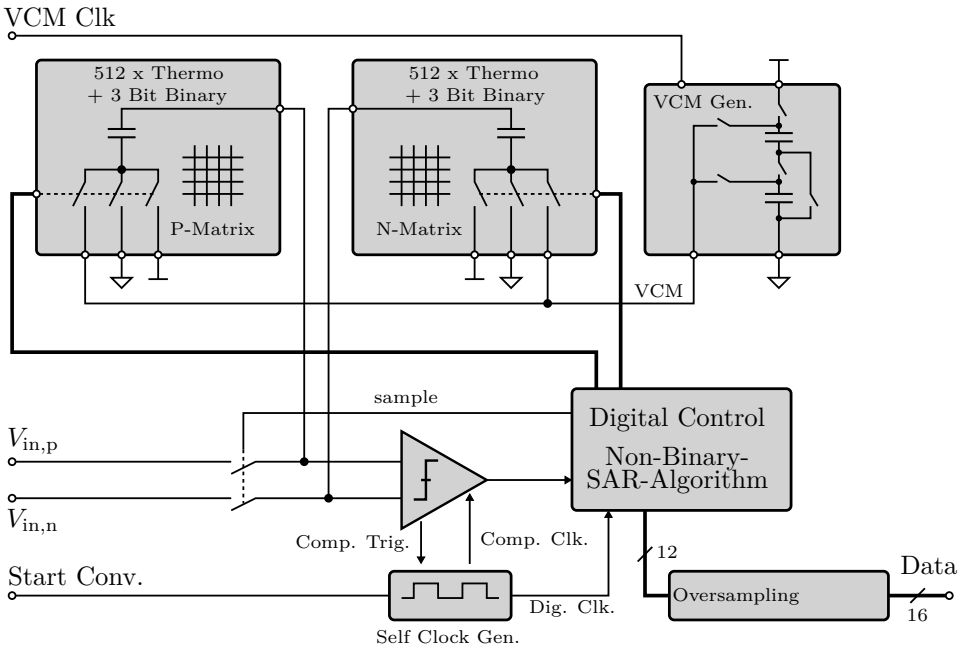



**Figure 1.7:** Block diagram of a non-binary 12-bit SAR ADC using self-clocking, integrated decimation filtering, and embedded switched-capacitor voltage divider for common-mode voltage generation. Image obtained from [9].

- Capacitive DAC `adc_array_matrix_12bit.gds` using a waffle-capacitor topology, whereas each cell consists of 8 unit-capacitors to reach high capacitance per area while the structure allows eased matching between binary fractions.
- DAC cells as small as possible to economize on area.
- Supply voltage $V_{\mathrm{DD}} = 1.8\,\mathrm{V}$ which is the SKY130 default supply voltage.
- 9 bit thermometer-code for inherently good differential linearity and 3 bit binary-code to get an area-efficient DAC with 12 bit.
- Control of matrix row- and column-decoder sequential or symmetric.
- Averaging of the 4 least significant SAR weights to reduce the influence of comparator noise.
- Up to 16 bit resolution with the usage of oversampling factor 256.
- Non-binary SAR algorithm to add redundancy for error correction.
- Dynamic latched comparator `adc_comparator_latch.gds` to save energy, with symmetric NOR gates to improve matching and reduce comparator offset.
- Embedded switched-capacitor $V_{\mathrm{CM}}$ generator.
- The top level is mixed-signal, it interfaces digital RTL with analog macrocells.
- Embedded common-mode voltage generator `adc_vcm_generator.gds`.
- Floorplan similar to Fig. 1.6 for efficient area and short interconnections.
- Apache 2.0 License to publish free and open-source (FOSS) silicon.

# Chapter 2

# Analog Design

This chapter covers the development and simulation of analog circuits and layouts. A decoupling capacitor, digital-to-analog converter (DAC), latched comparator and a switched-capacitor voltage divider have been implemented. The decoupling capacitor is using the MOSFET and the MIM capacitor structures, and they are used for power decoupling and also as the load for the designed latched dynamic comparator. The DAC matrix is assembled hierarchically using the DAC subcells core, binary, gate, drive and dummy cell. The DAC cell geometry allows arrangement in a matrix formation. A switched-capacitor voltage divider generates the reference voltage $V_{\mathrm{CM}}$. The individual layouts are arranged in dedicated function blocks, they are referred to as the analog macro cells which are the DAC, the voltage divider and the latched comparator.

## 2.1 Workflow

The generalized workflow of analog designs in this work starts with the specification of a desired circuit, the choice of the topology, and the definition of the parameter limits. The circuit diagrams are designed in the schematic editor `xschem` and are simulated using `ngspice/Xyce`. Furthermore, simulation results are plotted with `xschem/gaw3/gnuplot`. As the circuits were optimized based on simulation results, the layout was implemented using `magic`. Afterwards, the finished layouts are validated using a layout-vs-schematic (LVS) check using the tool `netgen` (`iic-lvs.sh` [13]), whereas the manufacturability is ensured with design-rule-checks (DRC) using `magic` and `klayout` (`iic-drc.sh` [13]). In addition, the RC circuit parasitics can be extracted in a `.spice` netlist format with `magic` (`iic-pex.sh` [13]) and simulated again, followed by revising the schematic and layout depending on the simulation results until the results meet the required performance characteristics.

The last step is the generation of `.gds` and `.lef` files using `magic`. For this step, a few properties need to be set beforehand to allow the following toolchain to handle the macrocell properly.

- `property LEFclass BLOCK`: This tells `OpenLane` that the extracted macrocell should be handled as a macrocell. The other option `CORE` is reserved for standard cells.
- `property FIXED_BBOX {XX XY YX YY}`: Sets the outer bounds of the macrocell.
- `property LEForigin {0 0}`: Origin coordinates of the macrocell.

In this work, the previously built `iic-osic-tools` (`github.com/iic-jku/iic-osic-tools`) Docker image was used for the whole design of this thesis. The image includes Linux with tools for analog, digital, and mixed-signal designs. Alternatively, the following list summarizes the current (02/2023) sources for the analog design toolset:

- SKY130: Open-PDKs installer (`github.com/RTimothyEdwards/open_pdks`).
- xschem: Electrical circuit simulator with internal waveform viewer and mixed-signal support (`github.com/StefanSchippers/xschem`).
- ngspice: Mixed-signal simulator (`github.com/ngspice/ngspice`).
- Xyce: High-performance parallel simulator (`xyce.sandia.gov`).
- gaw3: Waveform viewer (`github.com/StefanSchippers/xschem-gaw`).
- gnuplot: Graphing utility (`gnuplot.sourceforge.net`).
- magic VLSI layout tool: Analog layouts with pcell generation, DRC check, parasitic extraction and post-layout SPICE-file generation (`github.com/RTimothyEdwards/magic`).
- KLayout: GDSII editor with GDSII DRC check[1] (`klayout.de`).
- netgen: Netlist management system used for LVS check (`github.com/RTimothyEdwards/netgen`).

## 2.2 Decoupling Capacitor

For applications where decoupling with high capacitance per area is needed, like in a switched-capacitor voltage divider and a dynamic comparator, a decoupling capacitor design is necessary. Inspired by the capacitor cells in the raw design files of the SAR-ADC in [11], two differently sized capacitor cells were designed using the SKY130 PDK. Parameters and differences between the two final layouts are summarized in Table 2.1, and the layouts and layers are shown in Fig. 2.1.

The capacitor layout is designed with a focus on symmetry and placement in an array formation without violating DRC rules. The layouts combine two independent stacked capacitors, the bottom capacitor `nmoscap` is based on the MOSFET poly-diffusion capacity with additional finger-capacitor structures in the layers up to `metal2`. The top capacitor `mimcap` is using the dedicated MIMcap-layer between `metal3` and `metal4`. With these measures, a capacity per area of $7.2\,\mathrm{fF}/\mu\mathrm{m}^2$ has been reached.

[1] Note that `kLayout` GDS-DRC rules differ from `magic` DRC rules

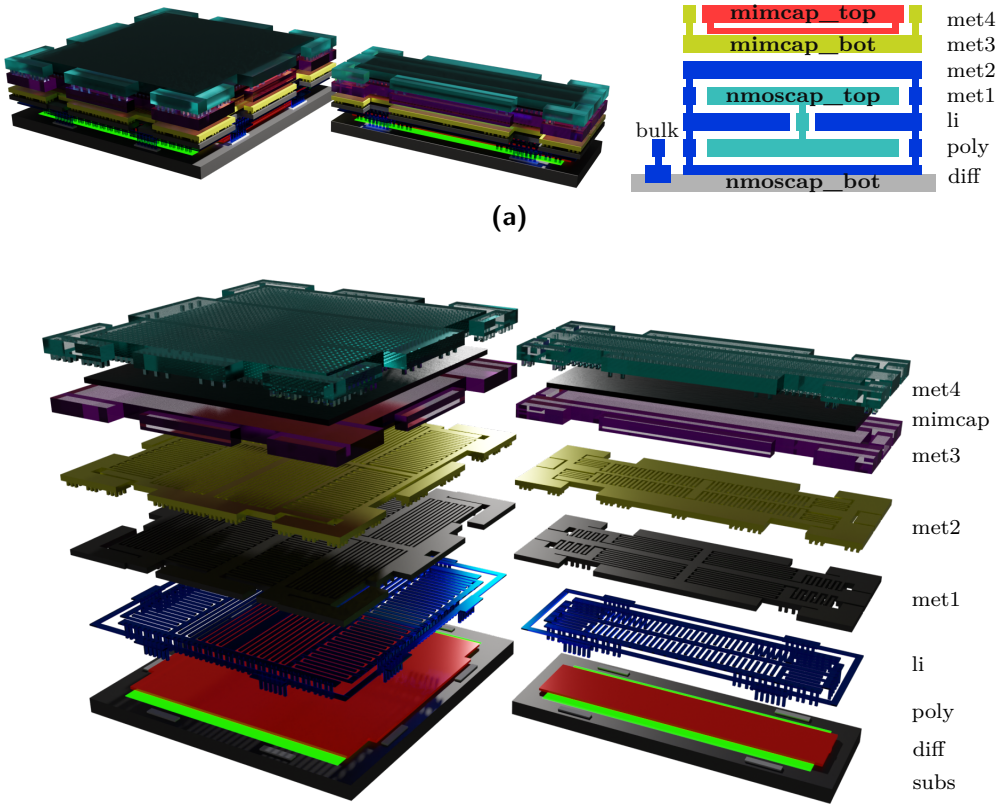



**Figure 2.1:** Rendered layouts of the decoupling cells. (a) `adc_noise_decoup_cell1.mag` (left) and `adc_noise_decoup_cell2.mag` (middle) with layer-stack diagram (right). (b) Exposure of the internal structures showing the implemented finger-capacitors. The top capacitor connectors are placed in the corner, the bottom capacitor connectors are at the center of the edge, therefore the capacitors can be placed in an array formation without further wiring.

**Table 2.1:** Decoupling capacitor parasitic extraction (PEX) results and calculated xschem component parameters.

| | `adc_noise_decoup_cell1` | `adc_noise_decoup_cell2` |
|---|---|---|
| Dimensions | 20.0 $\mu$m × 20.0 $\mu$m | 21.7 $\mu$m × 7.9 $\mu$m |
| nmoscap xschem | 1994 fF | 545 fF |
| nmoscap PEX | 253 fF | 72 fF |
| mimcap xschem | 604 fF | 202 fF |
| mimcap PEX | 30.91 fF | 14 fF |
| $C_{\text{nmoscap}}$ | 2247 fF | 617 fF |
| $C_{\text{mimcap}}$ | 635 fF | 216 fF |
| $C_{\text{total}}$ | 2882 fF | 833 fF |

The capacitance of the varactors (NMOS in nwell, accumulation mode) and the MIM capacitance have been directly extracted from the symbol attributes in `xschem`. Additional capacitances resulting from metal-insulator-metal structures were extracted with `magic` parasitic C extraction.

## 2.3 DAC Core Cell

In [10] a 10-bit SAR-ADC with DAC capacitor-matrix has been proposed based on a 130 nm process, the design is using capacitive cells arranged in an 8-bit matrix structure with additional 2 bit binary-coded cells. Each DAC cell has an embedded decoder for a row/column matrix structure allowing activation of the cells in a thermometer-code style for lower current consumption and inherently good differential linearity. Followed by this work, a 180 nm SAR-ADC design with 14 bit (10-bit thermometer-code, 4-bit binary) using a waffle-capacitor structure with 16 unit-capacitors per DAC core cell has been published [9]. The waffle-style structure of the capacitor ensures very low stray capacitance to other nodes than the bottom plate and good matching between the individual cells and the waffle structure allows a minimum unit-capacitance of only $C^0 = 0.37\,\text{fF}$.

To achieve competitive performance, the capacitor structures were compared with a focus on matching, capacitance, and area. The existing decoder circuit has been directly adapted with SKY130 minimum length MOSFETs. For charge injection compensation, semi-differential[2] wiring of the control wires was introduced for compensation of the charge injection through stray capacitances between the DAC control signals and the capacitor top plate. The result is a DAC capacitor cell with 8 unit-capacitors using a waffle structure and minimum unit-capacitance $C^0 = 0.447\,\text{fF}$ in the assembled 12-bit DAC matrix.

### 2.3.1 Comparison of the Capacitor-Structures

For evaluation of matching and capacitance between the binary- and the thermometer-code-capacitors, 8 different waffle- and finger-capacitor structures were drawn and compared with `magic`. Layer `m5` in the SKY130 layer stack shown in Fig. 1.1 was not used, since the design rules were disadvantageous for an area-efficient design (`m5` to `m5` distance, `via4` size). Layers `li` and `m1` are reserved for the decoder circuit wiring, which means the capacitor layout is drawn in the layers `m2`, `m3` and `m4`. One drawback of this layer stack is that the local interconnect layer has a sheet resistance of $12.8\,\Omega/\square$ [14], this could become the limiting factor regarding DAC speed since this layer has to be used for the column decoder. The chosen metal layers allow a better capacitor design, shielding is limited and therefore a solution with extensive use of `li` has been chosen.

The evaluated capacitor structures are shown in Fig. 2.2, they have been organized in a test structure with dummy cells as in Listing 2.1 to resemble a DAC capacitor-matrix. The commands shown in Listing 2.2 are a minimal example for parasitic C extraction using layout tool `magic`, the output is in `.spice` netlist format.

The extracted capacitances are shown in the resulting `.spice` file between `ctop` and `cbot`, the results are summarized in Table 2.2. Structures (a) and (b) show the best compromise regarding capacitance per area, matching, and parasitic capacitances.

[2] Differential layout and wiring, but single-ended evaluation of the signals.

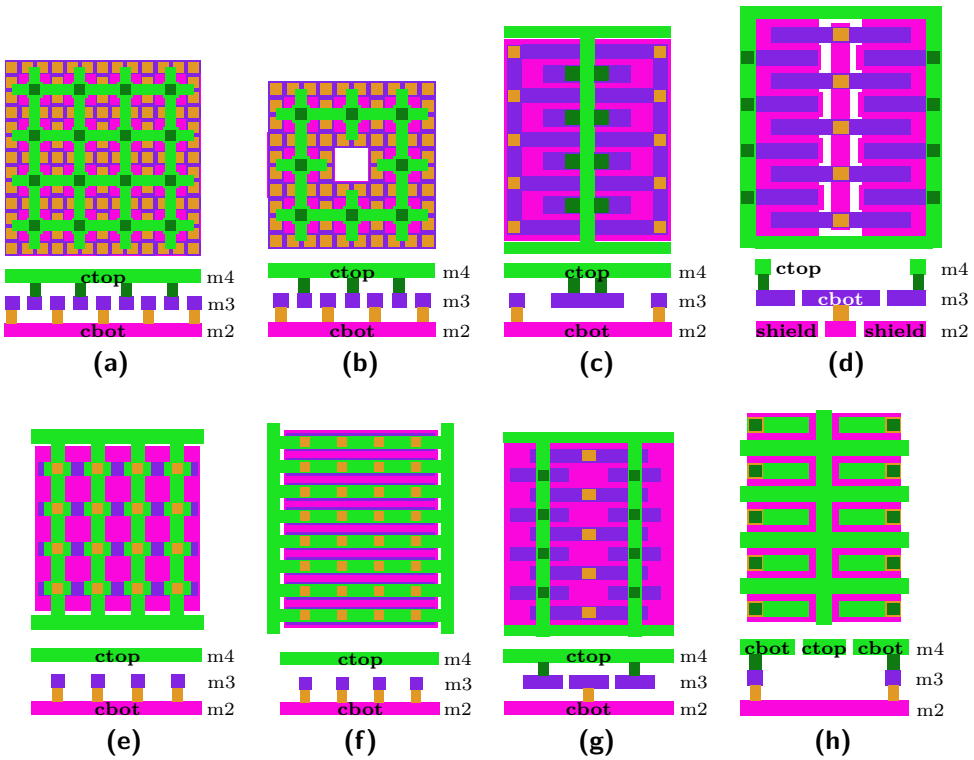



**Figure 2.2:** Layouts of the compared capacitor topologies. Topology (a) and (b) are waffle-capacitor structures, (d–h) are inspired by finger-capacitors, and (c) is a hybrid of both structures. Most structures have their main capacitance in layers m3-m3 (a–d,g), (e) and (f) use m3-m4 while (h) is located in m4-m4.

**Listing 2.1:** Teststructure for DAC cell capacitance extraction, exemplary for a DAC cell with 8 unit-capacitors.

```
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][8][8][8][ ][ ][8][8][8][ ][ ][8][8][8][ ][ ][8][8][8][ ]
[ ][8][8][8][ ][ ][8][8][8][ ][ ][8][8][8][ ][ ][8][8][8][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][4][ ][ ][ ][ ][2][ ][ ][ ][ ][1][ ][ ]

[ ] = Dummy
[1] = 1-unit cell
[2] = 2-unit cell
[4] = 4-unit cell
[8] = 8-unit cell
```

Setup (c) has the highest mismatch in the binary capacitor cells, which makes use of binary-coded DAC cells unreasonable, while structure (d) shows the best matching in the binary cells, but additional parasitics from ctop to the shield increase the gain error. The unit-capacitances in (e) and (f) are small and show high variance, therefore they are no candidates for this design. Structure (e) shows slightly more capacitance per area than (a) or (b), but the variance in the binary caps is also high. (f) shows a possible alternative with good matching and reasonable size, but based on these

**Listing 2.2:** A minimum example for the commands used for extraction of the parasitic capacitances using `magic`.

```
1  extract all
2  ext2spice lvs
3  ext2spice cthresh 0
4  ext2spice
```

**Table 2.2:** Extracted capacitances of the test structures.

| Test in Fig. 2.2 | (a) | (b) | (c) | (d) | (e) | (f) | (g) | (h) |
|---|---|---|---|---|---|---|---|---|
| Unitcaps $C^0$ per cell | 16 | 8 | 8 | 8 | 16 | 8 | 8 | 8 |
| Area per cell ($\mu$m$^2$) | 41 | 25 | 23 | 22 | 17 | 24 | 28 | 19 |
| Cellcap $C_{\text{cell}}$ (fF) | 9,18 | 4,77 | 0,61 | 3,12 | 0,74 | 1,07 | 5,55 | 4,97 |
| Unitcap $C^0_{16}$ (aF) | 572 | - | - | - | 46 | - | - | |
| Unitcap $C^0_8$ (aF) | 595 | 593 | 76 | 389 | 51 | 134 | 695 | 621 |
| Unitcap $C^0_4$ (aF) | 637 | 620 | - | 385 | 75 | 173 | 705 | 672 |
| Unitcap $C^0_2$ (aF) | 720 | 675 | - | 410 | 130 | 240 | 785 | 685 |
| Unitcap $C^0_1$ (aF) | 870 | 780 | 15 | 440 | 240 | 350 | 960 | 770 |

results the structures (a) and (b) (Waffle-capacitors) have been selected as the leading topology.

### 2.3.2 Waffle-Capacitor Sizing

The area of one DAC cell using a waffle structure can be optimized by overlapping the unified capacitors. A minimum area waffle-capacitor needs two `m3`-rails in one direction. The waffle-capacitor structure (a) in Fig. 2.2 shows that 16 overlapping waffle-capacitors need only 5 vertical `metal3` rails per direction instead of 8, therefore the effective capacitance per area is increased. This benefits the efficiency of the area with the introduction of binary-coded cells, which are a fraction of the overlapping cells. The total number of DAC cells decreases, therefore less minimum distances between DAC cells affect the total area. At some point, the benefit of this technique vanishes because dummy cells at the border of the matrix grow in size with the level of binary cell abstraction.

Considering that a capacitive DAC matrix with $N$ bit needs a predefined amount $2^N$ of unified capacitors, while it is desired to use the maximum advantage of binary capacitors, a minimized value for the matrix area can be found. The calculations in Table 2.3 reveal there is a tradeoff between DAC resolution and the number of unit-capacitors per cell where the area efficiency is optimized, and the minimum area per DAC resolution is highlighted with bold text. Based on these calculations, the waffle-capacitor topology (b) in Fig. 2.2 with 8 unified capacitors per cell has been selected in this work for a 12-bit DAC matrix.

**Table 2.3:** Approximation of the matrix area for a wafflecap-topology using binary cells and dummy cells, whereas the minimum area per DAC resolution is marked as bold text.

| | | Overlapping Capacitors per DAC-cell | | | | | |
|---|---|---|---|---|---|---|---|
| | Area in $\mu m^2$ | 4 | 8 | 16 | 64 | 256 | 1024 |
| DAC Resolution | 4 bit | **256** | 302 | 376 | | | |
| | 5 bit | **384** | 403 | 501 | | | |
| | 6 bit | **576** | 605 | 668 | 1344 | | |
| | 7 bit | 960 | **907** | 1002 | 1792 | | |
| | 8 bit | 1600 | 1512 | **1502** | 2389 | 5072 | |
| | 9 bit | 2880 | 2520 | **2504** | 3584 | 6763 | |
| | 10 bit | 5184 | 4536 | **4173** | 5376 | 9017 | |
| | 11 bit | 9792 | 8165 | **7512** | 8960 | 13526 | 26260 |
| | 12 bit | 18496 | 15423 | **13521** | 14933 | 20289 | 35014 |
| | 13 bit | 35904 | 29132 | **25540** | 26879 | 33815 | 52521 |
| | 14 bit | 69696 | 56550 | **48242** | 48382 | 56359 | 78781 |
| | 15 bit | 137280 | 109773 | 93646 | **91389** | 101446 | 131302 |
| | 16 bit | 270400 | 216219 | 181783 | **172624** | 182602 | 218837 |
| | 17 bit | 536640 | 425887 | 358057 | **335093** | 344916 | 393906 |
| | 18 bit | 1065024 | 845221 | 705264 | **650475** | 651507 | 709031 |

### 2.3.3 Cell Decoder

A matrix row and column decoder [9] with thermometer-style cell activation has been implemented into the DAC cells, the circuit is shown in Fig. 2.3. Two inversely activated, complementary gates function as an analog multiplexer, this multiplexer connects the bottom plate of the capacitor to either the common-mode voltage $V_{\mathrm{CM}}$ if signal `sample` = 1.8 V and to `vdrv` if `sample` = 0 V. The voltage of `vdrv` is $V_{\mathrm{DD}}$ if the row and column control signals ($\overline{\texttt{row}}$ and $\overline{\texttt{col}}$) of an individual cell in the matrix are both active, simultaneously `vdrv` in a full row can be set to $V_{\mathrm{DD}}$ using signal $\overline{\texttt{rowon}}$. If none of these conditions are met, then `vdrv` is tied to $V_{\mathrm{SS}}$. This logic allows the DAC cells to be activated using thermometer coding in a matrix structure. The MOSFETs have been sized to use minimum length ($L = 0.15\,\mu$m, $W_{\mathrm{nmos}} = 0.42\,\mu$m, $W_{\mathrm{pmos}} = 0.80\,\mu$m) and regular $V_{th}$ doping.

### 2.3.4 Implementation

With `magic` a DAC core cell layout was implemented using the waffle-capacitor structure with 8 unit-capacitors and the decoder circuit with a focus on minimizing the area as shown in Fig. 2.4, and Fig. 2.5. The cell occupies a total area of $25.2\,\mu m^2$. Wires $\overline{\texttt{row}}$, `sample`, $V_{\mathrm{DD}}$ and $V_{\mathrm{SS}}$ have been implemented on the horizontal `m1` layer with ports oriented in the west and east of the layout, this allows seamless arraying of the DAC cell in `magic` by executing the command `array X Y`. For the same reason, the column control signals are arranged vertically. The common-mode reference-voltage $V_{\mathrm{CM}}$ is distributed with horizontal rails.

To compensate for the charge injection through coupling between row and column control wires to the capacitor top plate, semi-differential signals have been integrated into the layout. Post-layout simulation of the assembled DAC revealed that parasitic

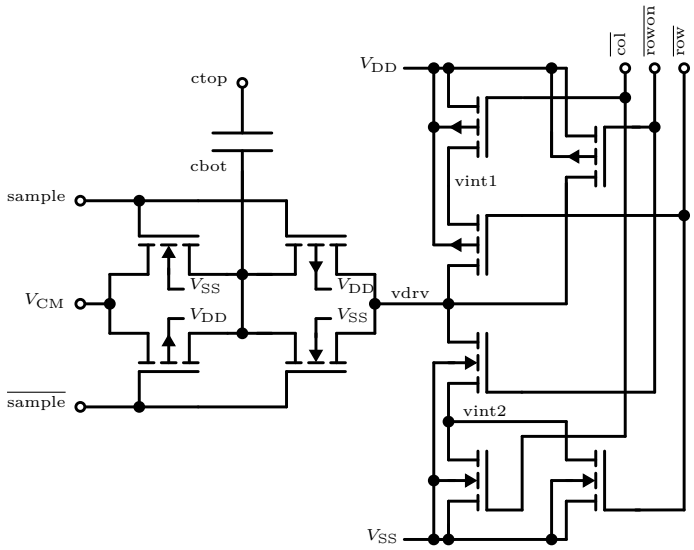



**Figure 2.3:** Row/column decoder circuit-diagram [9] integrated into the DAC cells for thermometer-code activation in a matrix row/column formation.

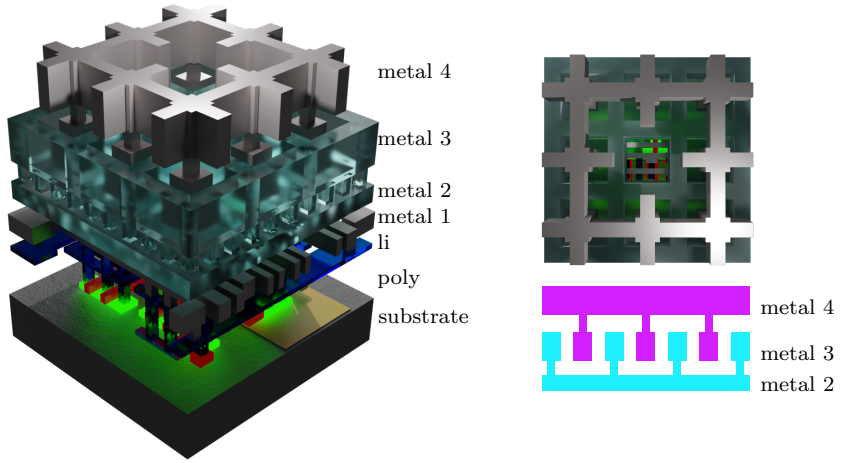
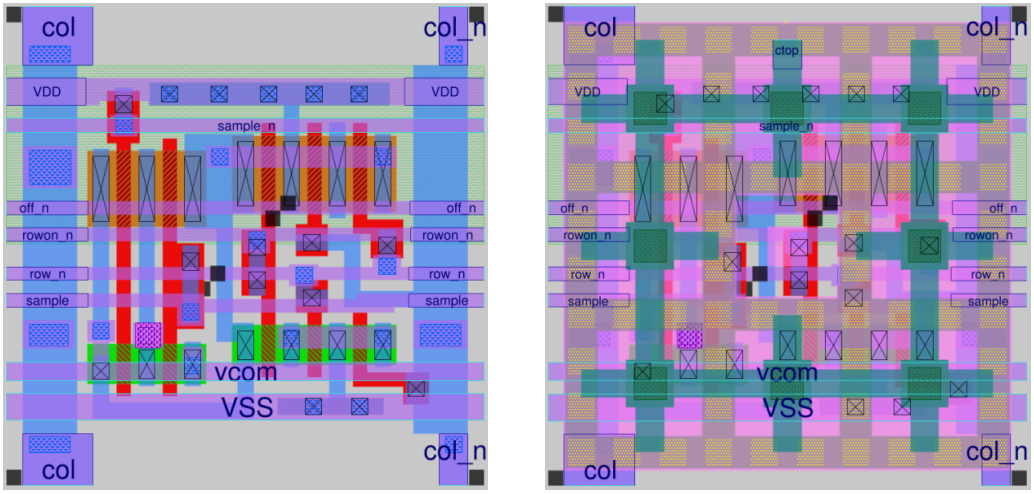



**Figure 2.4:** Rendered DAC core cell `adc_array_wafflecap_8.mag` with 8 unit-capacitors on top and schematic view of the MIM-capacitor.

capacitances between $\overline{\texttt{col}}$ and `ctop` have a significant impact on the DAC output voltage. For example, one of the extracted capacitances in the assembled 12-bit DAC between the capacitor top plate and one of the column control wires shows a value of 2.04 fF, if the row in a 12-bit matrix is changed then all 32 column signals ($32 \times 2.04$ fF parasitic capacitance) switch their states simultaneously, this leads to a performance-degrading output voltage jump which cannot be ignored. More details regarding this topic are shown in Sec. 2.8.2.

The signal wire `col` is differential to $\overline{\texttt{col}}$ and has no LVS-relevant function, its purpose is to pull charge from `ctop` when $\overline{\texttt{col}}$ injects charge and vice versa. Note that the physical layouts of `col` and $\overline{\texttt{col}}$ are not symmetrical, the dimensions have been chosen based on `magic` parasitic extraction after the DAC matrix has been assembled. To decode the rows, three states are used: Row on, row off, and row selected. These




**(a)** Layout of the decoder circuit with hidden layers above `m1`

**(b)** Complete layout with decoder circuit and capacitor on top

**Figure 2.5:** `Magic` layout `adc_array_wafflecap_8.mag` of (a) the decoder circuit (metal layers `m2`, `m3` and `m4` hidden), (b) full layout of the DAC core cell with 8 waffle-style capacitors on top and decoder circuit below.

states can be mapped with two signal wires, $\overline{\texttt{row}}$ and $\overline{\texttt{rowon}}$, but in this bit-efficient configuration, the average charge induced from these control signals at the capacitor top plate is not constant. To compensate for the charge injection, the redundant wire $\overline{\texttt{rowoff}}$ was added. This additional signal wire is also non-functional, but its additional parasitic capacitance can be used to compensate for induced charge. This setting allows the mapping of all three states with compensated charge injection.

## 2.4 DAC Binary Cell

For the realization of the 3-bit binary DAC code, in addition to the 9-bit thermometer-code, binary cells have been implemented. With a waffle-capacitor topology made of 8 unified capacitor cells (equivalent to the binary value `1000` where one unified capacitor maps to the LSB) matching is simplified and only the appropriate number of capacitor cells need to be connected in the binary cells [9]. Based on this idea, DAC cells with 4 unit-capacitors `adc_array_wafflecap_4.mag` mapping the value `0100`, 2 unit-capacitors `adc_array_wafflecap_2.mag` for `0010` and 1 unit-capacitor `adc_array_wafflecap_1.mag` for the LSB `0001` (Fig. 2.7) have been designed. The binary cells are not addressed with row and column signals in the DAC matrix, instead, a separated binary cell enable signal $\overline{\texttt{bin\_en}}$ has been added to override the row and column control signals as shown in Fig. 2.6. For better matching between the cells, no further optimizations have been applied on the circuit, e.g. removal of 4 redundant MOSFETs in the row and column decoder logic.

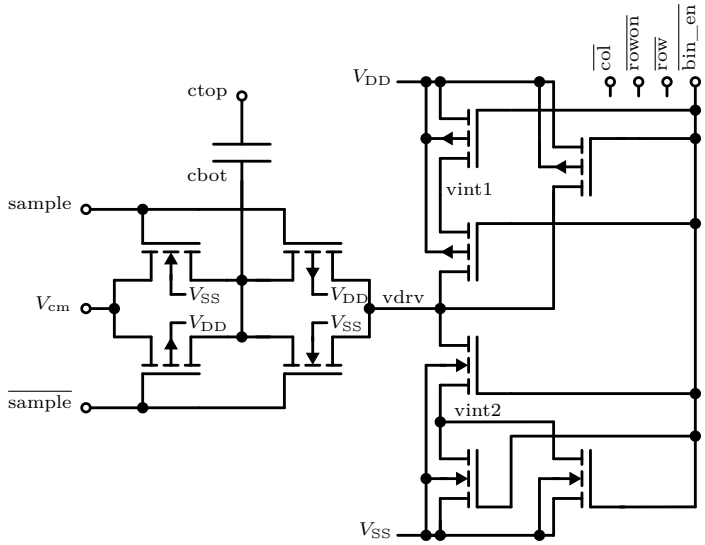



**Figure 2.6:** Decoder circuit of the binary capacitor DAC cell.

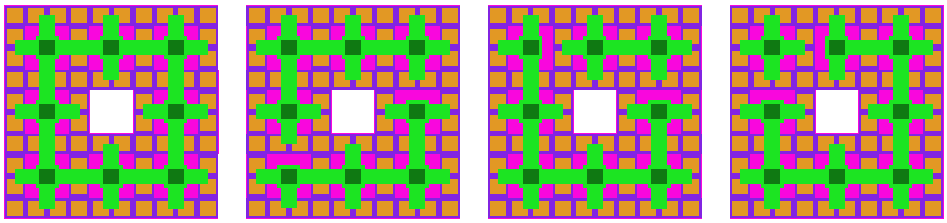



**Figure 2.7:** Implementation of the DAC binary cell capacitor layouts. (a) Layout top-view of the core cell waffle-capacitor with 8 unit-capacitors for reference. (b) Binary cell with 4 unit-capacitors. (c) Binary cell with 2 unit-capacitors. (d) Binary cell with 1 unit-capacitor.

## 2.5 DAC Gate Cell

For the sample-and-hold functionality of the analog-to-digital converter, a complementary switch is needed. In sample mode, `ctop` should be connected to the input signal, in hold mode, those nets need to be high-ohmic to be disconnected from the input signal. The goal is to build a gate cell layout with an S & H switch in the form factor of the DAC core cell.

### 2.5.1 Dimensioning of the Sample-and-Hold Switch

To ensure sampling speed at the maximum sample rate the switch needs proper dimensioning. While increasing the $W/L$ would increase speed as it lowers the $R_{on}$ of the switch, simultaneously the precision is being degraded due to distortion, which can occur if the switches do not turn off simultaneously, channel charge injection, leakage currents, and clock feedthrough [15]. The objective is to find a tradeoff where the switch is as big as necessary to reach the specified sampling speed, and simultaneously as small as possible to get sufficiently high off resistance and low charge injection.

The switch has to face three different challenges:

1. While sampling: If the voltage at the switch input $V_{\text{in}}$ jumps from $V_{\text{SS}}$ to $V_{\text{DD}}$, then the on-resistance must be small enough so that the voltage difference between the switch output and its input $|V_{\text{out}} - V_{\text{in}}|$ is less than 1 LSB after the shortest possible timespan for sampling.
2. While holding: If the input changes, then the output is not allowed to change for more than 1 LSB during the longest possible hold time.
3. The voltage at the switch output should stay constant if the mode is changed from sample to hold, therefore charge injection must be evaluated.

#### 2.5.1.1 Determination of the maximum Switch On-Resistance $R_{\text{on}}$

During the sampling phase, a voltage jump at the switch input from $V_{\text{SS}}$ to $V_{\text{DD}}$ and the fastest sampling speed determines the maximum allowed switch resistance. The designed 12-bit DAC using the SKY130 technology has an LSB voltage of

$$V_{\text{LSB}} = \frac{V_{\text{DD}} - V_{\text{SS}}}{2^N - 1} = \frac{1.8\,\text{V}}{4095} = 440\,\mu\text{V} \tag{2.1}$$

The shortest sampling phase is determined by the fastest self-clocked loop setting. Assuming only 2 of 3 delay components are required with a delay of $t_{\text{delay}} = 5\,\text{ns}$ per cell, the clock period for the digital clock generated by 2 delay cells with 2 clock-loop cycles results in $T_{\text{clk}} = 4 \cdot 5\,\text{ns} = 20\,\text{ns}$. The shortest sampling time is therefore $\Delta t = T_{\text{clk}} = 20\,\text{ns}$. The switch must be able to charge the ADC top plate of the capacitor from $U_C(t_0) = 0\,\text{V}$ to $U_C(t_0 + \Delta t) > (V_{\text{DD}} - V_{\text{LSB}}) = 1.8\,\text{V}\frac{4094}{4095}$ in this timespan.

The time-dependent voltage of a capacitor is calculated:

$$U_C(t) = U_0 \left(1 - \exp\left(-\frac{t}{RC}\right)\right) \tag{2.2}$$

Rearranging the variables and applying a logarithmic function leads to the equation:

$$\ln\left(\frac{U_0}{U_0 - U_C(\Delta t)}\right) = \frac{\Delta t}{RC} \tag{2.3}$$

Based on the extracted capacitance-topology test results, the capacitance of a single DAC cell is expected to be $C_{\text{unit}} = 4.77\,\text{fF}$, the extrapolated capacitance for a 12-bit matrix is, therefore, $C_{\text{matrix}} = 2^N \cdot C_{\text{unit}} = 2.44\,\text{pF}$. Inserting the values $U_C(\Delta t) > 1.8\,\text{V}\frac{4094}{4095}$, $U_0 = 1.8\,\text{V}$ and $C_{\text{matrix}}$ in Eq. 2.3 and rearranging the variables to express $R_{\text{on}}$ leads to the limiting value for the on-resistance:

$$R_{\text{on}} < \frac{\Delta t}{C \cdot \ln(4095)} = \frac{20\,\text{ns}}{2.44\,\text{pF} \cdot \ln(4095)} = 985\,\Omega \tag{2.4}$$

#### 2.5.1.2 Determination of the allowed leakage current $I_{\text{leakage,max}}$

The non-binary digital logic is designed to use 15 clock cycles for the evaluation of 12 non-averaged bits and 2 additional clock cycles for sampling and oversampling control. 4 bit LSB conversion results can be averaged which uses up to 30 additional clock cycles per bit. Using the maximum amount of LSB averaging cycles, a single conversion takes $N_{\text{cycles}} = N_{\text{control}} + N_{\text{MSB}} + N_{\text{LSB}} = 2 + 11 + 4 \cdot 31 = 137$ clock cycles. The allowed voltage change at the switch output $V_{\text{out}}$ in this timespan is the LSB voltage $\Delta V_{\text{out}} < V_{\text{LSB}} = 440\,\mu\text{V}$.

The delay cell can be set in a range of 0 ns to 155 ns, the longest clock period in the clock generator is therefore $\Delta t = 6 \cdot 155\,\text{ns} = 930\,\text{ns}$ per clock cycle. The hold time in the slowest conversion mode with maximized delay setting in the delay chain is determined by:

$$T_{\text{hold}} = 137\text{cycles} \cdot 930\,\text{ns/cycle} = 127.4\,\mu\text{s} \tag{2.5}$$

The slowest sample rate $f_{\text{s,min}}$ is calculated by inversion of the longest hold time.

$$f_{\text{s,min}} = \frac{1}{T_{\text{hold}}} = 7849\,\text{samples/s} \tag{2.6}$$

The allowed leakage current for these values is calculated by inserting the previously approximated value for the matrix capacitance $C_{\text{matrix}} = 2.44\,\text{pF}$:

$$I_{\text{leakage}} = C_{\text{matrix}} \frac{\Delta V_{C_{\text{matrix}}}}{\Delta t} < I_{\text{leakage,max}} \tag{2.7}$$

$$I_{\text{leakage,max}} = C_{\text{matrix}} \frac{V_{\text{LSB}}}{T_{\text{hold}}} = 2.44\,\text{pF} \frac{440\,\mu\text{V}}{127.4\,\mu\text{s}} = 8.43\,\text{pA} \tag{2.8}$$

#### 2.5.1.3 Evaluation of Simulations

According to the previous calculations, the allowed on-resistance is $R_{\text{on}}^{\text{max}} = 985\,\Omega$ and the leakage-current is limited to $I_{\text{leakage}}^{\text{max}} = 8.43\,\text{pA}$. The complementary switch has been simulated with low, mid and high $V_{\text{threshold}}$ MOSFET types: `lvt`, `nvt` and `hvt` using the setup shown in Fig. 2.11.

Evaluation of the simulated leakage currents $I_{\text{leakage}}$ reveals that the `lvt` type MOSFETs tend to escalate currents as shown in Fig. 2.8, and do not meet the design specifications. Therefore, the `lvt` type is excluded from the selection. Also, the `nvt` type with the shortest gate length is excluded because the leakage current exceeds the limits. The `hvt` types show overall higher leakage currents compared to the relevant nominal types if the gate length is increased, therefore the best choice at corner `tt` is the complementary switch with nominal-type MOSFET `nvt` and increased gate length.

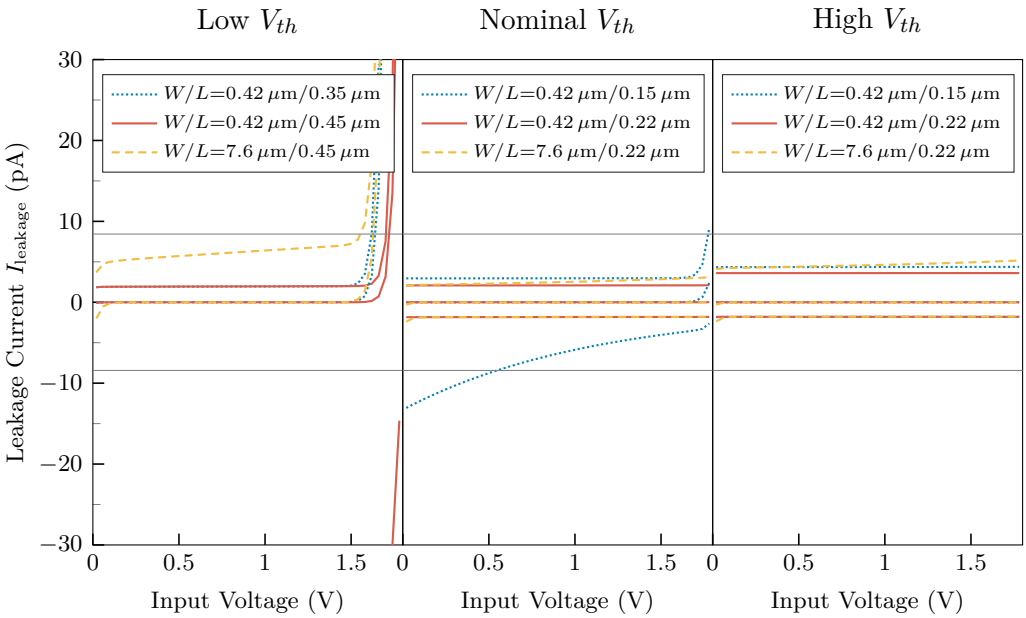



**Figure 2.8:** Simulation results of the complementary switch leakage current while the switch is off. DC sweep of the input voltage from 0 V to 1.8 V (x-axis). Simulations with different MOSFET widths $W$ and lengths $L$ are differentiated by color and line type. Plots from left to right: low $V_{th}$, nominal $V_{th}$, high $V_{th}$. The output voltage was set to 0 V, 0.9 V and 1.8 V (waveform overlays in plots with the same color). The calculated limit for leakage current is displayed with a gray line.

When judging the $R_{\text{on}}$ curves in Fig. 2.9, then the performance is best with `lvt` and worst with `hvt`. The `lvt` type with $W = 7.6\,\mu$m is exceeding the limit $R_{\text{on}}^{\text{max}}$ in transition, but the mean stays below the limit. The `nvt` and `hvt` types exceed the limit clearly, but overall the `nvt` reaches a better performance than the `hvt` type at identical dimensions. The on-resistance at $W = 7.6\,\mu$m and $L = 0.22\,\mu$m is not ideal yet, but the result looks like a good compromise between size and performance. The advantage of these dimensions, they nearly use all of the area available beneath a DAC cell, therefore the gate can be integrated into the ADC matrix structure.

The final simulation results with the chosen dimensions, including a variation of corner and temperature, can be seen in Fig. 2.10. The slow corner might tend to have too high on-resistance $R_{\text{on}}$, while in the fast corner the leakage current $I_{\text{leakage}}$ seems to be a limiting factor, especially at the edge of the input voltage range. The simulation results show a good compromise between both values for the targeted corner `tt`.

#### 2.5.1.4 Schematic

Because the `lvt` and the shortest-length `nvt` types have been excluded, and performance regarding on-resistance becomes better with lower $V_{th}$, the chosen MOSFETs for the complementary switch are nominal $V_{th}$ type with dimensions $W = 7.6\,\mu$m and $L = 0.22\,\mu$m (M1, M2). Although the problems with charge injection and clock feedthrough might already be relieved due to the usage of differential signals [15], charge injection compensating dummy fets M3 and M4 ($W = 3.8\,\mu$m, $L = 0.22\,\mu$m)

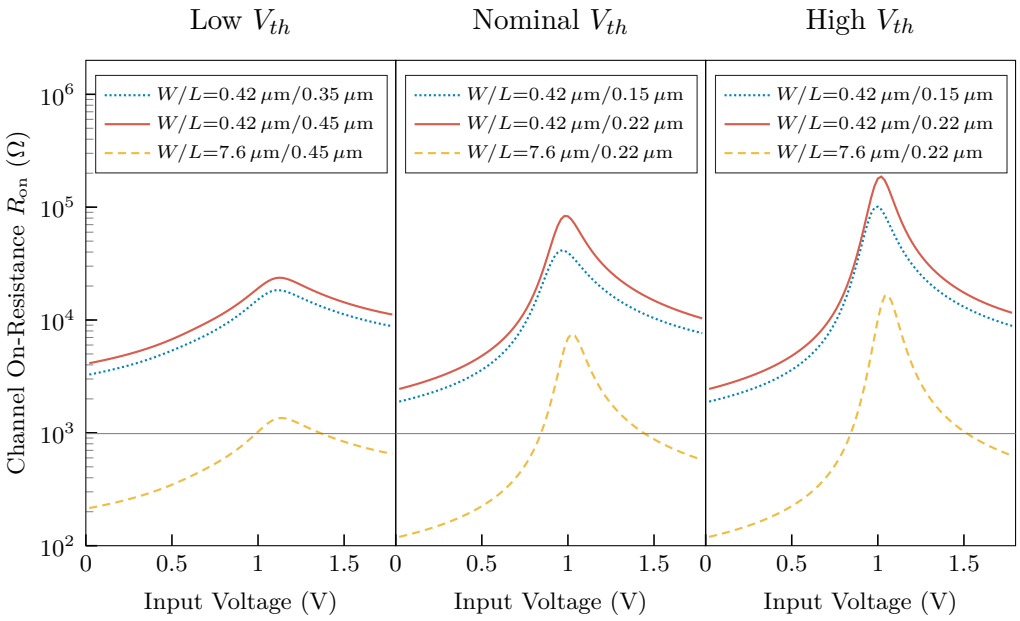



**Figure 2.9:** Simulation results for on-resistance of the complementary switch. DC-sweep of the input voltage from 0 V to 1.8 V (x-axis), the voltage between drain and source has been set to $V_{DS} = 1\,mV$ and the switch is on. Color differentiates between width $W$ and length $L$ further specified in the legends. The left plot and the right plot show a comparison between low-, nominal-, and high-$V_{th}$ MOSFET types (`lvt`, `nvt`, `hvt`).

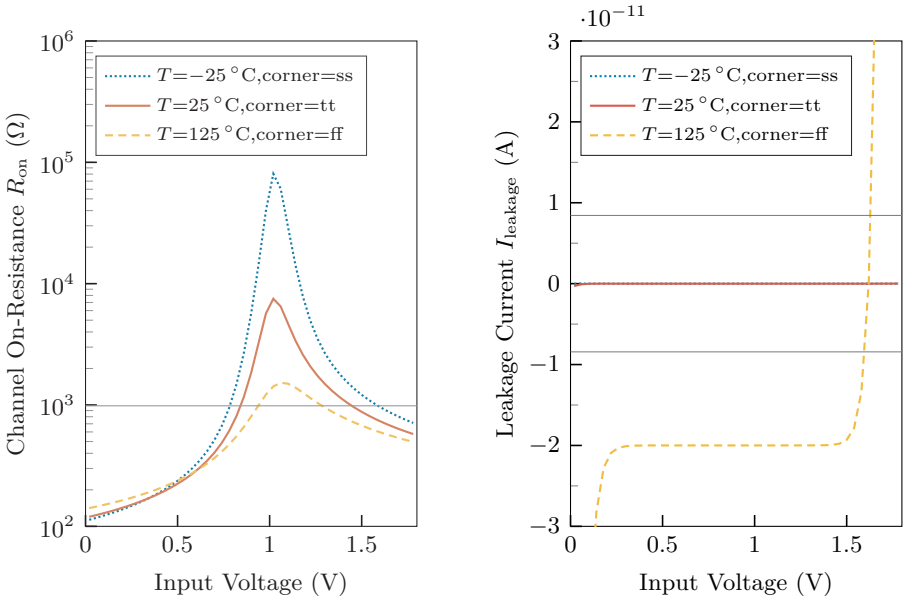



**Figure 2.10:** Complementary switch with $W = 7.6\,\mu m$ and $L = 0.22\,\mu m$ at different corners, DC-sweep of the input voltage from 0 V to 1.8 V (x-axis). (a) On-resistance $R_{on}$, the drain to source voltage has been fixed to $V_{DS} = 1\,mV$. (b) Leakage current $I_{leakage}$, the output of the switch has been set to $V_{out} = 0.9\,V$ since this voltage is expected to be regulated against $V_{CM} = 0.9\,V$ by the SAR algorithm.

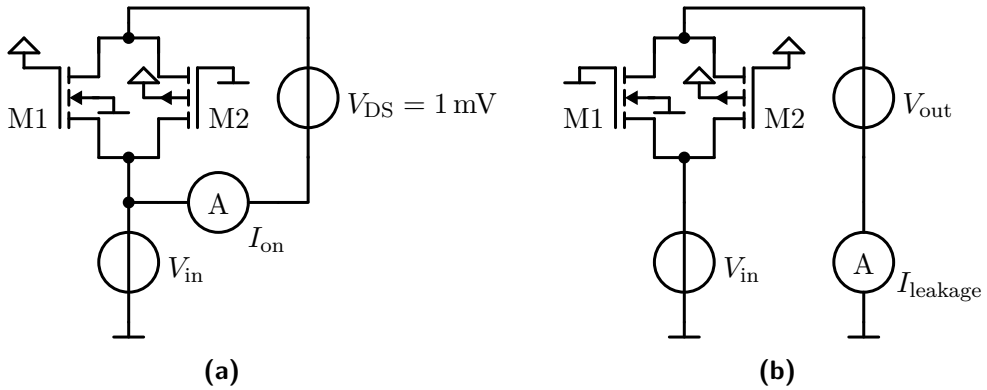



**Figure 2.11:** Testbenches for (a) the on-resistance and (b) leakage-current. $R_{on}$ is calculated with $V_{DS}/I_{on}$.

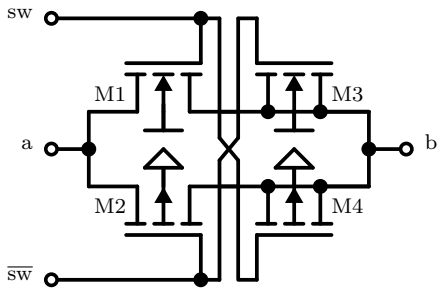



**Figure 2.12:** Schematic of the implemented gate-switch for the sample and hold functionality with in/outputs `a` and `b`, and the complementary switch control signal `sw`.

have been integrated into the design (Fig. 2.12) on the DAC-capacitor side of the switch (M3, M4).

#### 2.5.1.5 Bootstrapped Switch

In the complementary switch shown in Fig. 2.12, the NMOS and PMOS gate voltages $V_g$ are limited to the values $V_{DD}$ or $V_{SS}$ while the source voltages $V_s$ follow the input voltage. Therefore, the gate-source voltages $V_{gs} = V_g - V_s$, and consequently the $R_{on}$, are a function of the input voltage. The dependence in the complementary switch of $R_{on}$ to the input voltage is shown in Fig. 2.9. This relation limits the performance of the complementary switch in the input voltage range between $V_{SS}$ and $V_{DD}$. High $R_{on}$ can be adjusted by a change of the MOSFET widths, but the increased drain-junction capacitance can distort the signal as a result of adding a nonlinear capacitance to the output. The bootstrapped switch proposed in [16] aims to minimize $R_{on}$ variation at large input voltage swings by forcing a constant gate-source voltage $V_{gs}$.

In this work, the use of a bootstrapped switch has been dismissed because of the following reasons: The $R_{on}$ and $I_{leakage}$ design targets can be met with a simple complementary gate. The gate voltages exceed 1.8 V in the bootstrapped switch, this would have made the usage of MOSFETs with higher voltage ratings necessary. Simulations have shown an increased leakage current in high-voltage MOSFETs compared to nominal-voltage MOSFETs degrading performance at lower data rates, which is disadvantageous for biomedical sensor applications where low data rates are used

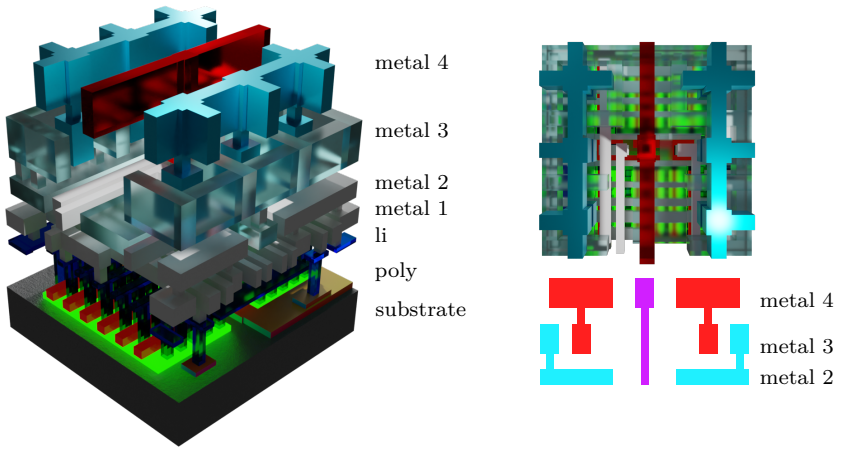



**Figure 2.13:** 3D rendered gate cell `adc_array_wafflecap_gate.mag` using the DAC cell geometry.

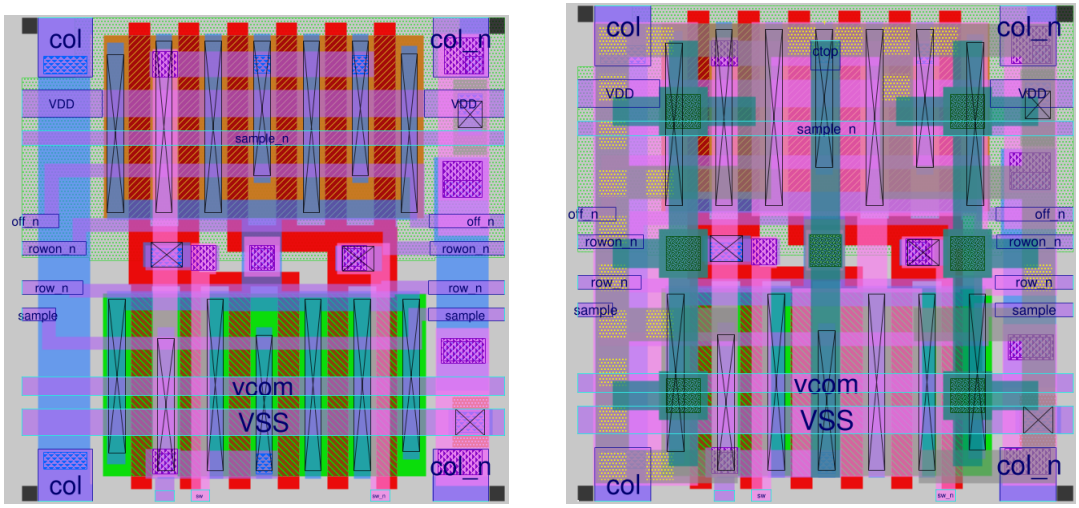
to decrease power consumption. Additionally, the switch can not be always-on, an additional clock cycle is needed to prepare the bootstrapped sampling switch. Furthermore, the additional bootstrapping circuit would have complicated the integration of the sample-and-hold switch into one of the DAC matrix cell layouts.

### 2.5.2 Implementation

The previously designed DAC core cell with waffle-capacitor as shown in Fig. 2.4 has been reused and adapted to the sample-and-hold functionality as shown in Fig. 2.14 and Fig. 2.13. The complementary switch fits neatly under the existing layout, the decoder circuit has been removed in this cell. Row and column control signals are gated to neighboring cells, which limits the usage of `met1` for the complementary switch routing. The structure from the waffle-capacitors has been preserved as much as possible to partly mimic a DAC dummy cell. Because of that reason, layer `met2` (which is reserved for the capacitor bottom plate in the capacitor DAC cell) has also been used for wiring. Caution has been kept to route the top capacitor connections as short as possible with increased awareness about parasitic capacitances.

## 2.6 DAC Drive Cell

The DAC drive cell has the purpose to generate the complementary sample signals for the DAC matrix rows. Simulations have shown that these sample signals must change their state simultaneously, otherwise, the pass gates as seen in Fig. 2.3 open a low-impedance path between $V_{\mathrm{CM}}$ and vdrv. To ensure precise state changes of the sample signals, the complementary clock generation circuit as shown in Fig. 2.15 has been embedded in the DAC drive cell using the same geometry as the previously designed DAC cell shown in Fig. 2.4. Capacitor structures have been preserved to allow the replacement of a DAC dummy cell with the DAC drive cell. The integrated clock generation circuit uses MOSFET dimensions $W_{\mathrm{NMOS}} = 0.6\,\mu\mathrm{m}$, $W_{\mathrm{PMOS}} = 1.2\,\mu\mathrm{m}$ and $L = 0.15\,\mu\mathrm{m}$ with nominal $V_{th}$ type doping.




**(a)** Layout of the DAC gate cell circuit with hidden top-layers.

**(b)** Complete layout of the DAC gate cell.

**Figure 2.14:** `Magic` layout `adc_array_wafflecap_gate.mag` of (a) the complementary switch (metal layers `m4`, `m3` and capacitor bottom-plate related `m2` hidden), (b) complete layout of the DAC gate cell.

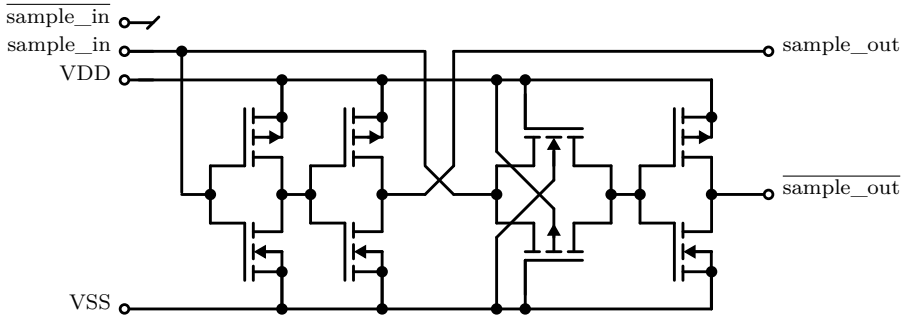



**Figure 2.15:** Schematic of the DAC drive cell complementary sample signal generator for moderate precision [15]. The pass gate duplicates the delay of one inverter.

## 2.7 Latched Comparator

An energy-efficient two-stage comparator [17] with additional trigger-output signaling the end of conversions, and latched output using symmetric NOR-gates [9], for an improved design and layout symmetry, have been implemented in this component.

### 2.7.1 Comparator Design

The comparator schematic is shown in Fig. 2.16 and the chosen MOSFET types are further specified in Table 2.4. Capacitors C1 and C2 are realized with the MOS capacitor part of cell `adc_noise_decoup_cell2` with 617 fF each. The cell combines a stacked MOSFET-capacitor and MIM-capacitor, the MIM structures are tied to $V_{SS}$ and used as shielding. The capacitor value of 617 fF has been chosen on basis of the implemented values in [11], which used a noise-driven design plan for ultra-low-power SAR-ADC designs. X1 and X2 are simple buffer circuits using two inverter pairs (X1/2.MN1, X1/2.MP1) and (X1/2.MN2, X1/2.MP2).

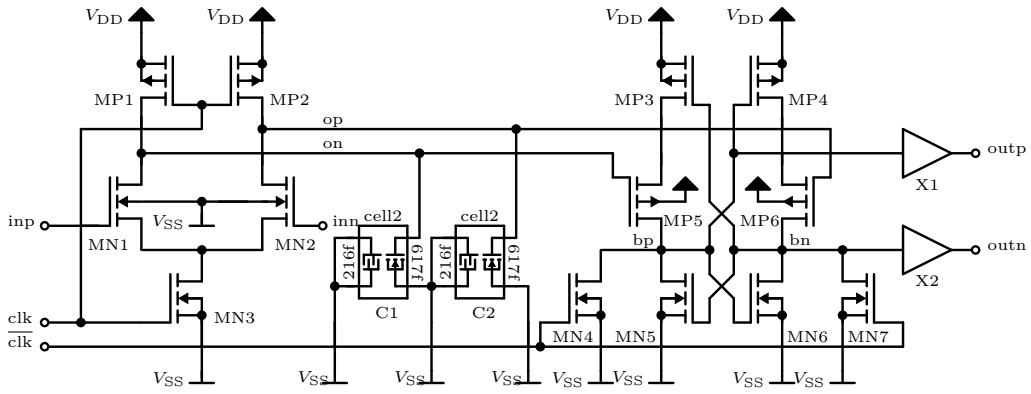



**Figure 2.16:** Schematic of the implemented dynamic comparator [9]. MOSFET types and dimensions are shown in Table 2.4.

In reset mode (`clk`=low) MOSFETs MP1 and MP2 are conducting while MN3 is off, and capacitors C1 and C2 are charged against the supply voltage $V_{DD}$. Simultaneously MN4 and MN7 are switched on to pull `bn` and `bp` to $V_{SS}$, this presets the output stage for the comparator decision.

After transitioning to conversion mode (`clk`=high) MOSFETs MN4 and MN7 turn off and the output stage is released into an unstable equilibrium. MOSFETs MP1 and MP2 are also turned off, MN3 is now on and the capacitors C1 and C2 discharge through MN1 and MN2. Voltage difference at the input signals (`inp-inn`) results in a difference in the capacitor discharge rates. The faster-discharging capacitor forces the output stage to trip from the unstable equilibrium into the desired stable state, which reflects the comparison result at the outputs.

### 2.7.2 Latched Comparator with Symmetric NAND

The latched comparator [9] combines the previously described comparator as shown in Fig. 2.16 with the NOR-based latch and trigger signal generation. The clock signal is buffered and inverted through a two-stage inverter chain ($W_{NMOS} = 0.42\,\mu$m, $W_{PMOS} = 0.84\,\mu$m, $L = 0.15\,\mu$m) to generate both the inverted clock signal $\overline{\texttt{clk}}$ and the buffered regular signal `clk`.

While the comparator X3 shown in Fig. 2.17 is in reset (`clk`=low), both comparator-outputs `outp` and `outn` are being pulled low, therefore output $\overline{\texttt{comp\_trig}}$ of NOR X4 is high. The NOR S-R latch (X5, X6) is stable, output value `q` is latched.

When the comparator decision is triggered (`clk`=high), then the comparator will trip into a stable state where `outp` is either high or low, `outn` becomes the inverse of `outp`. Since `outp` and `outn` are now different, $\overline{\texttt{comp\_trig}}$ will change to low signaling a finished conversion and the S-R latch adapts its output state to match the conversion result.

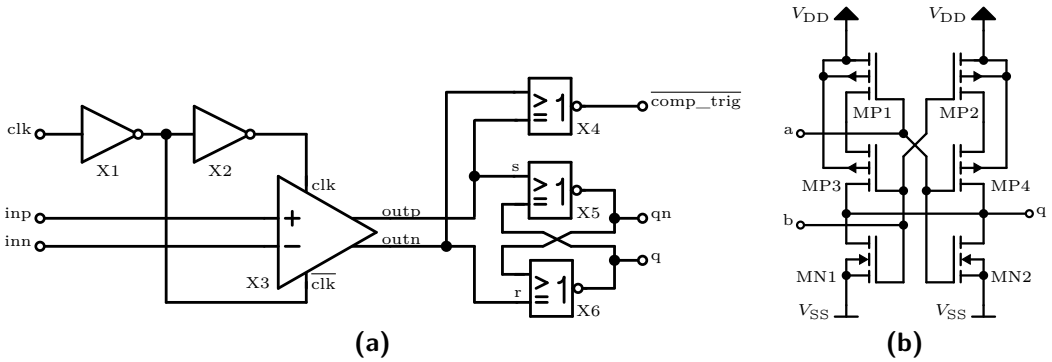



**Figure 2.17:** (a) Latched dynamic comparator [9] using NOR gates for the `comp_trig` signal and RS latch, the comparator circuit is shown in Fig. 2.16. (b) Symmetric NOR gate [9] for better symmetry in design and layout compared to a regular NOR gate. Corresponding MOSFET types and dimensions are shown in Table 2.5.

**Table 2.4:** Comparator NMOS/PMOS dimensions.

| Refdes. | Model | L ($\mu$m) | W ($\mu$m) |
|---|---|---|---|
| MN1/2 | nomimal $V_{th}$ | $1 \times 0.15$ | $4 \times 2$ |
| MN3 | nomimal $V_{th}$ | $1 \times 0.15$ | $8 \times 0.5$ |
| MN4/5/6/7 | nomimal $V_{th}$ | $1 \times 0.15$ | $1 \times 2$ |
| MP1/2 | nomimal $V_{th}$ | $1 \times 0.15$ | $4 \times 0.5$ |
| MP3/4/5/6 | nomimal $V_{th}$ | $1 \times 0.15$ | $4 \times 1$ |
| C1/2.MN1 | nomimal $V_{th}$ | $1 \times 3.9$ | $1 \times 18.4$ |
| X1/2.MN1 | nomimal $V_{th}$ | $1 \times 0.15$ | $1 \times 0.5$ |
| X1/2.MP1 | nomimal $V_{th}$ | $1 \times 0.15$ | $1 \times 1$ |
| X1/2.MN2 | nomimal $V_{th}$ | $1 \times 0.15$ | $2 \times 0.5$ |
| X1/2.MP2 | nomimal $V_{th}$ | $1 \times 0.15$ | $2 \times 1$ |

### 2.7.3 Implementation

The layout has been done using `magic`, and a rendered image of the result is shown in Fig. 2.18. Comparator input and output stages are shown in Fig. 2.16 and the differential wires (`inp`+`inn`, `op`+`on`) have been placed and routed with increased attention on symmetry, especially an unsymmetric input stage would significantly decrease the precision of the comparator. In the design process, the parasitic wire capacitances to the substrate have been extracted using the `iic-pex.sh` script [13] in the C decoupled mode for a rough check of matching.

The inputs (`inp`, `inn`, `clk`) are located on the left border of the macro, the outputs (`latch_q`, `latch_qn`, $\overline{\texttt{comp\_trig}}$) on the right edge. Differential signals have been routed as close as possible to reduce error by differential-mode interference, common-mode interference is principally already compensated by the usage of differential signals. Dummy MOSFET structures have been added whenever possible to further improve the symmetry of the comparator design. The resulting layout dimension is $28.3\,\mu\text{m} \times 28.0\,\mu\text{m}$.

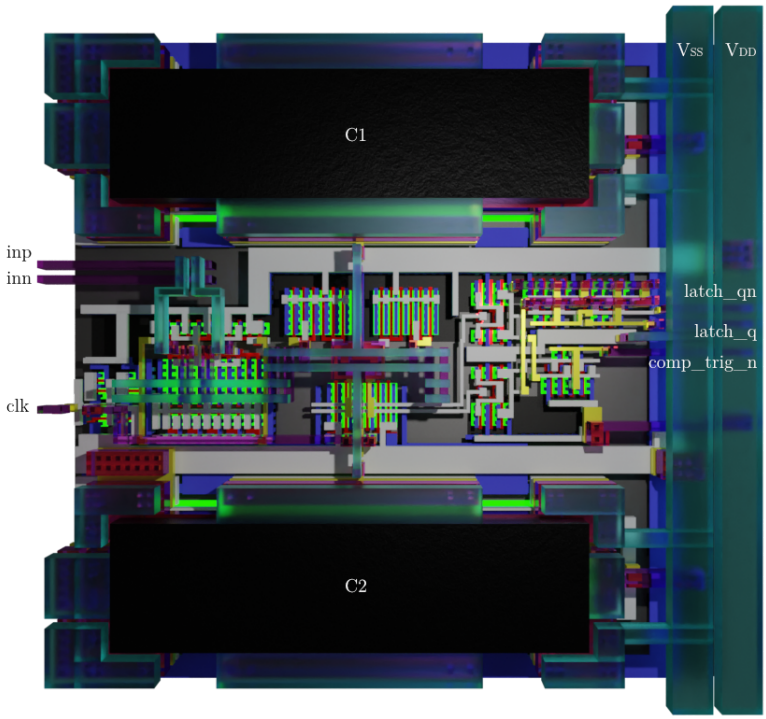



**Figure 2.18:** Rendered image of the latched comparator `adc_comp_latch.mag`. The comparator input stage is located on the left, and the top and bottom are occupied by the rectangular capacitor-cell `adc_noise_decoup_cell2.mag`. The second comparator stage is located in the middle of the layout, nearby two buffers placed on the right followed by 3 NOR gates of the S-R latch and $\overline{\texttt{comp\_trig}}$ signal generator. Power is supposed to be connected with horizontal `metal5` power rails to the vertical `metal4` power rails on the right.

**Table 2.5:** Symmetric NOR latch NMOS/PMOS dimensions.

| Refdes. | Model | L ($\mu$m) | W ($\mu$m) |
|---|---|---|---|
| MN1/2 | nomimal $V_{th}$ | $1 \times 0.15$ | $1 \times 0.42$ |
| MP1/2/3/4 | nomimal $V_{th}$ | $1 \times 0.15$ | $1 \times 0.80$ |

## 2.8 Capacitive DAC

With the previously designed DAC cells (core `adc_array_wafflecap_8.mag`, dummy `adc_array_wafflecap_dummy.mag`, drive `adc_array_wafflecap_drv.mag`, gate `adc_array_wafflecap_gate.mag`, binary `adc_array_wafflecap_4/2/1.mag`) the DAC-matrix layout can be assembled as shown in Fig. 2.19 and simulated.

### 2.8.1 Matrix Assembly

For a physical capacitive DAC resolution of 12 bit, $2^{12} = 4096$ unit-capacitors $C^0$ are necessary for the DAC matrix which is equivalent to $16 \times 32$ core cells with 8 unit-capacitors $C^0$ each. The matrix is surrounded by dummy cells, which add 2 rows and columns resulting in a total matrix dimension of $18 \times 34$. With 3-bit binary-coded cells (1, 2 and 4 unit-capacitors $C^0$ per cell) and 9-bit thermometer-code DAC core cells, 5 dummy cells on the bottom are replaced with 4 binary cells (2x1, 1x2, 1x4) in the center and one gate cell in the corner. The drive cells in the first column are modified dummy cells with a complementary clock generator circuit, they are used to

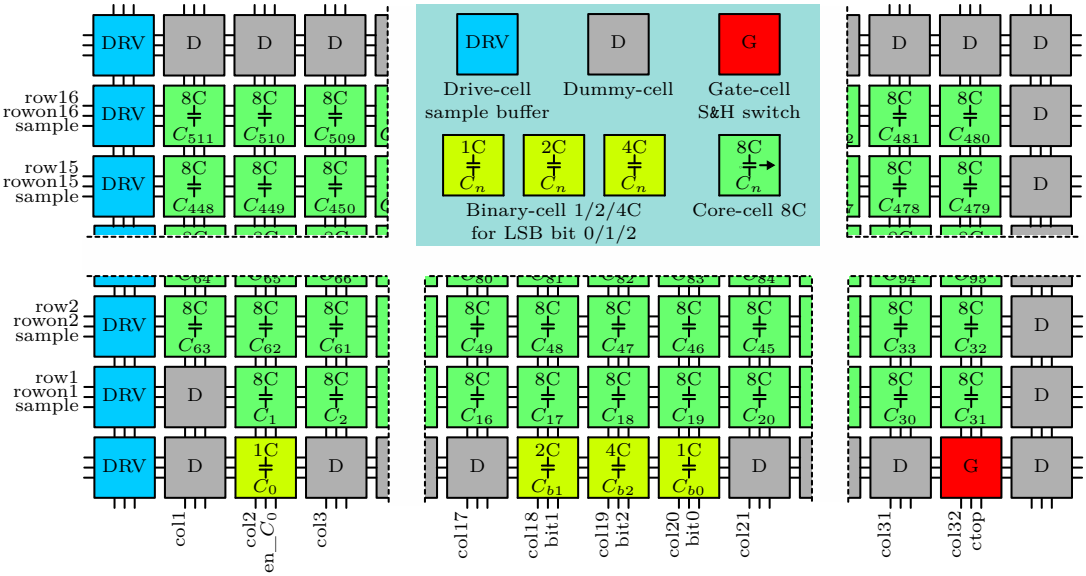



**Figure 2.19:** DAC-Matrix block diagram showing the arrangement of DAC cells.

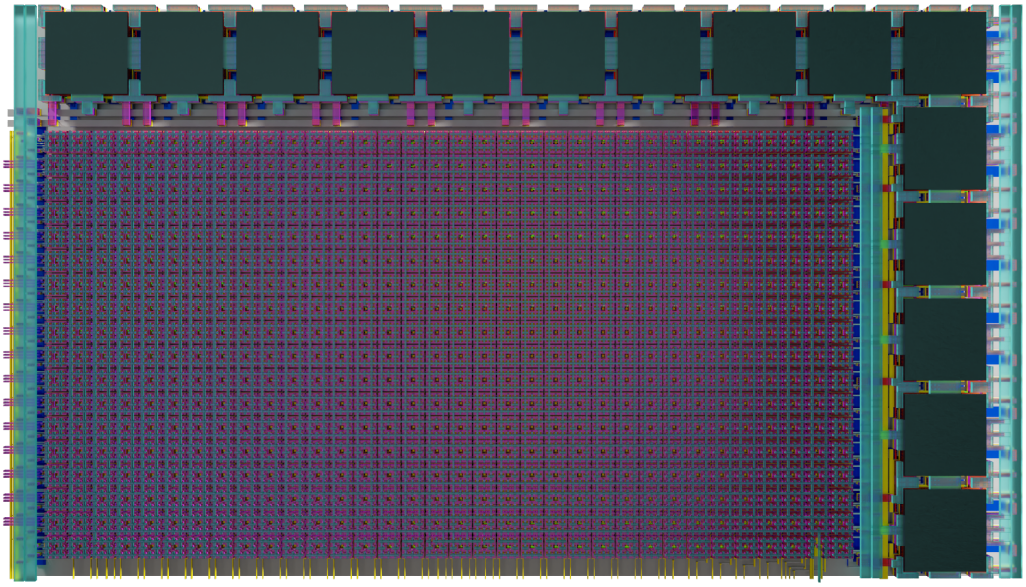


**Figure 2.20:** The final layout of the DAC Matrix. Decoupling capacitors are located on the north and east side, the signal interface is south and west. Power is connected via vertical power rails.

buffer and generate the symmetric sample signals once per row. For implementing the function of binary cells, one core cell with 8 unified capacitors is split into its binary fractions while the first core cell is changed to a dummy cell. This results in a final compilation of $2 \cdot 1C^0 + 1 \cdot 2C^0 + 1 \cdot 4C^0 + (2^9 - 1) \cdot 8C^0 = 4096C^0$ active unit-capacitors.

With Fig. 1.6 as a template for the floorplan, the ADC matrix macrocell has been designed to interface the signal wires on the south and west. Power is connected over vertical `metal4` power rails using the horizontal `metal5` power distribution network. Decoupling capacitors (`adc_noise_decoup_cell1.mag`) have been placed north and east of the macrocell.

**Listing 2.3:** DAC matrix I/O interface

```
module adc_array_matrix (
   inout VDD,              // 1.8 V supply
   inout VSS,              // Ground
   inout vcm,              // 0.9 V Common Mode Voltage
   input sample,sample_n, // set sample or hold mode
   input [15:0] row_n,
   input [15:0] rowon_n,
   input [15:0] rowoff_n,
   input [31:0] col_n,
   input [31:0] col,
   input [2:0] en_bit_n,      // enable binary capacitors
   input en_C0_n,             // enable first capacitor C0
   input sw, sw_n, analog_in, // switch, connect analog_in->ctop
   output ctop);
endmodule
```

This results in the DAC I/O interface from Listing 2.3. $V_{\mathrm{DD}}$, $V_{\mathrm{SS}}$ and $V_{\mathrm{CM}}$ are power-nets. Wire `sample(_n)` sets the DAC to sampling or hold mode, `sw(_n)` is the S & H switch for `analog_in`, note that it is important to ensure that the gate-switch is fully closed before `sample(_n)` activates hold-mode, otherwise, the capacitor bottom plate voltages will change before the top plate is isolated from the analog input. Bus signals `row_n`, `rowon_n` and `rowoff_n` are the row-enable signals, similarly, `col_n` and `col` enable the columns. `en_C0_n` switches the first binary capacitor $C_0$, `en_bit_n` the other binary capacitors. `ctop` is the analog connection to the capacitor top plate.

### 2.8.2 Semi-Differential Charge Compensation

The median capacitance of the column control wires `col_n[31:0]` to the top capacitor plate `ctop` using the extracted parasitic capacitances as seen in Listing 2.4 is $C_{\mathrm{col\_n}} = 2.02\,\mathrm{fF}$ per wire. It has been shown by simulation that 32 column control wires changing their state simultaneously[3] induce significant charge injection to `ctop` through a parasitic capacitance of $31 \times C_{\mathrm{col\_n}} = 32 \times 2.02\,\mathrm{fF} = 64.64\,\mathrm{fF}$. Compared to the DAC capacitance between the top and bottom plates $C_{\mathrm{matrix}} = 1.83\,\mathrm{pF}$, which has been calculated in Equation 2.14, the occurring voltage swing $\Delta V_{\mathrm{ctop}}$ induced by the parasitic capacitance of 32 column control wires can be calculated with the formula for a capacitive voltage divider:

$$\Delta V_{\mathrm{ctop}} = V_{\mathrm{DD}} \frac{C_{\mathrm{parasitics}}}{C_{\mathrm{parasitics}} + C_{\mathrm{ctop}}} = 1.8\,\mathrm{V} \frac{64.64\,\mathrm{fF}}{64.64\,\mathrm{fF} + 1.83\,\mathrm{pF}} = 61\,\mathrm{mV} \tag{2.9}$$

With 16 rows in the DAC matrix and using the simulation results for the maximum and minimum DAC voltages $V_{\mathrm{DAC}}^{\mathrm{max}}$ and $V_{\mathrm{DAC}}^{\mathrm{min}}$ as seen in Figure 2.22, the calculated DAC voltage swing per row $\Delta V_{\mathrm{row}}$ is:

$$\Delta V_{\mathrm{row}} = \frac{V_{\mathrm{DAC}}^{\mathrm{max}} - V_{\mathrm{DAC}}^{\mathrm{min}}}{\mathrm{rows}} = \frac{1.627\,\mathrm{V} - 0.1672\,\mathrm{V}}{16\,\mathrm{rows}} = 91.2\,\mathrm{mV/row} \tag{2.10}$$

[3] This happens if the row-col decoder steps through the 16 rows.

**Listing 2.4:** DAC matrix `.spice` file snippets of the extracted capacitances at `ctop`

```
...
a_31078_3134# ctop 3.57fF
a_16018_17190# ctop 3.39fF
a_26058_2130# ctop 3.39fF
a_10998_16186# ctop 3.57fF
a_34090_5142# ctop 3.42fF
a_5978_15182# ctop 3.58fF
a_29070_4138# ctop 3.58fF
a_24050_3134# ctop 3.57fF
a_8990_17190# ctop 3.39fF
a_19030_2130# ctop 3.50fF
...
vcm ctop 33.18fF
VDD ctop 92.86fF
...
col_n[24] ctop 2.02fF
ctop col[7] 1.98fF
ctop col[18] 1.98fF
ctop col[29] 1.98fF
col_n[3] ctop 2.02fF
col_n[14] ctop 2.02fF
col_n[25] ctop 2.02fF
...
cbot_bin1 ctopp 0.65fF
cbot_bin2 ctopp 1.01fF
cbot_bin4 ctopp 1.70fF
```

Changing the voltages on all 32 capacitor bottom plates from $V_{SS}$ to $V_{DD}$ within one row is expected to induce a voltage change of 91.2 mV on ctop. Simultaneously the `col_n` signals are switched from $V_{DD}$ to $V_{SS}$, and the parasitic voltage divider reduces the voltage on `ctop` by −61 mV. The voltage swing within a row is therefore reduced from [0 mV to 91.2 mV] to [0 mV to 30.2 mV]. This is a reduction of 66.9%!

$$\frac{\Delta V_{\text{parasitics}}}{\Delta V_{\text{row}}} = \frac{61\,\text{mV}}{91.2\,\text{mV}} = 66.9\% \tag{2.11}$$

Note for the transition to the next row all 32 `col_n` wires change simultaneously from $V_{SS}$ to $V_{DD}$, which means it is possible to see the parasitic contribution of 61 mV in the DAC output voltage in a single LSB code-step.

As a countermeasure, semi-differential[4] wires have been added in the DAC cell layouts to compensate for the parasitic charge injection as shown in Fig. 2.21. Matching of the semi-differential control signals to `ctop` can be evaluated by comparing the capacitances of `col[31:0]` to `col_n[31:0]` in the extracted netlist. The `col[31:0]` to `ctop` capacitances have a median value of $C_{\text{col}} = 1.98\,\text{fF}$ per wire while the `col_n[31:0]` to `ctop` capacitances have $C_{\text{col\_n}} = 2.02\,\text{fF}$ as exemplary shown in Listing 2.4. Simulation of the DAC transfer curve in Fig. 2.22 proves that the charge

[4]Semi-differential because the added wires have no LVS relevant function, their only purpose is to induce controlled parasitic capacitance.

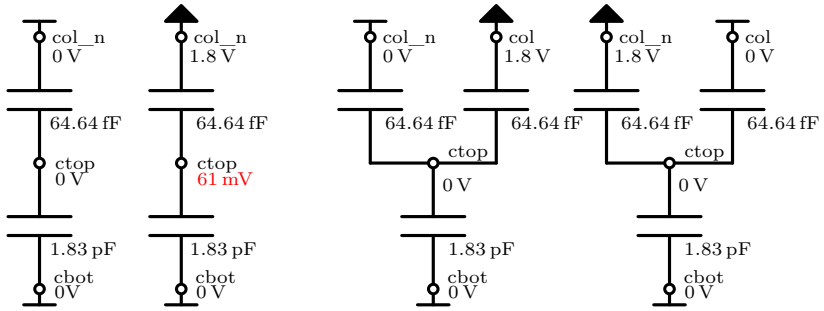



**Figure 2.21:** The functional principle of the semi-differential charge compensation. If no charge compensation is performed then a state change of `col_n` will interfere with `ctop` (left). With differential charge compensation (right) the voltage on `ctop` stays constant.

injection compensation does work in the theoretical model, and the DAC transfer curve linearity is restored.

### 2.8.3 Simulation Results

A capacitive parasitic extraction has been performed on the finalized layout with `magic` and the voltage transfer curve of the pre-layout and post-layout DAC has been simulated with `ngspice`, simulation results are shown in Fig. 2.22 using testbench `adc_row_col_decoder_tb.sch`. The row and column driver is a digital XSPICE netlist containing a row & column decoder. The decoder is further described in the digital design section of this work in Sec. 4, in essence, it converts a digital 12-bit value to thermometer-code row and column signals for the DAC matrix. The transient simulation starts in sampling mode where both DAC capacitor top-plates are charged to $V_{\text{top}} = 0.9\,\text{V}$. The S & H switch is closed, followed by the sampling signal. The input data generation starts at 0 and increments the DAC input data every 500 ns up to 4095.

If the transfer function tends to follow a simple linear model $y = ax + m$, then the gain is the slope and m is the offset of the transfer waveform function, a mismatch in $a$ and $m$ is called gain- and offset-error. Since the transfer function is not perfectly linear the gain error is usually measured between code zero and the endpoint or points close to zero and maximum and specified in terms of %FSR (full-scale range) [18]. Simulations showed that the post-layout DAC output range values from 0.1672 V to 1.627 V. With a full-scale range of 0 V to 1.8 V the gain error $e_{\text{gain}}$ can be calculated as:

$$e_{\text{gain}} = \frac{1.8\,\text{V} - 1.627\,\text{V} + 0.1672\,\text{V}}{1.8\,\text{V}} = 19\%\texttt{FSR} \tag{2.12}$$

Gain error is caused by capacitive voltage dividers between `cbot - ctop - <net>`, it is therefore encouraged to minimize capacitance from `ctop` to other nets than the corresponding `cbot`. Capacitances `"VDD ctop 92.86fF"` and `"vcm ctop 33.18fF"` are suspected to mainly cause the gain error in the DAC transfer curve. To not further increase $e_{\text{gain}}$ top capacitor structures and the bottom capacitor of the dummy cells have been left floating, this is expected to be a tradeoff between parasitic matching and gain error.

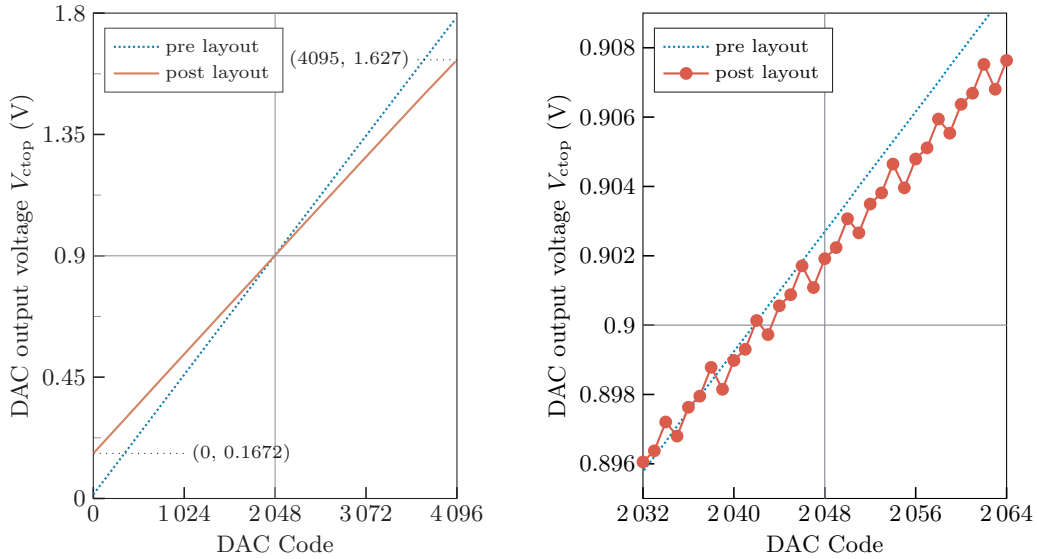



**Figure 2.22:** ADC DAC matrix transfer curves of the ideal simulation model and the post layout `.spice` netlist, the data vector sweeps from value 0 to 4095. (a) shows the full transfer curve, (b) is zoomed in to the center showing a slight mismatch in the LSB bit translation.

**Table 2.6:** Extracted capacitances of the binary cells $C^4$, $C^2$ and $C^1$. Mismatch calculation versus the median extracted capacitance of the core cells $C^8$.

| $C^8 = 3.58\,\text{fF}$ | expected | extracted | mismatch |
|---|---|---|---|
| $C^4$ | 1.79 fF | 1.70 fF | −5.0% |
| $C^2$ | 895 aF | 1010 aF | +12.8% |
| $C^1$ | 447 aF | 650 aF | +45% |

By examining the `.spice` netlist of the post layout C extraction, the final capacitance of the waffle-capacitor structure can be found by filtering for keyword `ctop`. Code snippets are summarized in Listing 2.4. There are 511 $(16 \times 32 - 1)$ capacitances between `ctop` and nets named `a_<number>_<number>` above 3.24 fF, these are the capacitances between `ctop` and `cbot` of the individual DAC core cells. 420 capacitances ($14 \times 30$ DAC core cells in the center) are found to be in the range 3.56 fF to 3.58 fF with a median capacitive value of 3.58 fF. Based on the median value of core cell capacitors the desired unit-capacitance $C^0$ and the total capacitance $C_{\text{matrix}}$ can be calculated:

$$C^0 = \frac{3.58\,\text{fF}}{8} = 448\,\text{aF} \tag{2.13}$$

$$C_{\text{matrix}} = 4096 \times \frac{3.58\,\text{fF}}{8} = 1.83\,\text{pF} \tag{2.14}$$

The desired binary cell capacitances have been compared to the extracted binary capacitor values in Table 2.6.

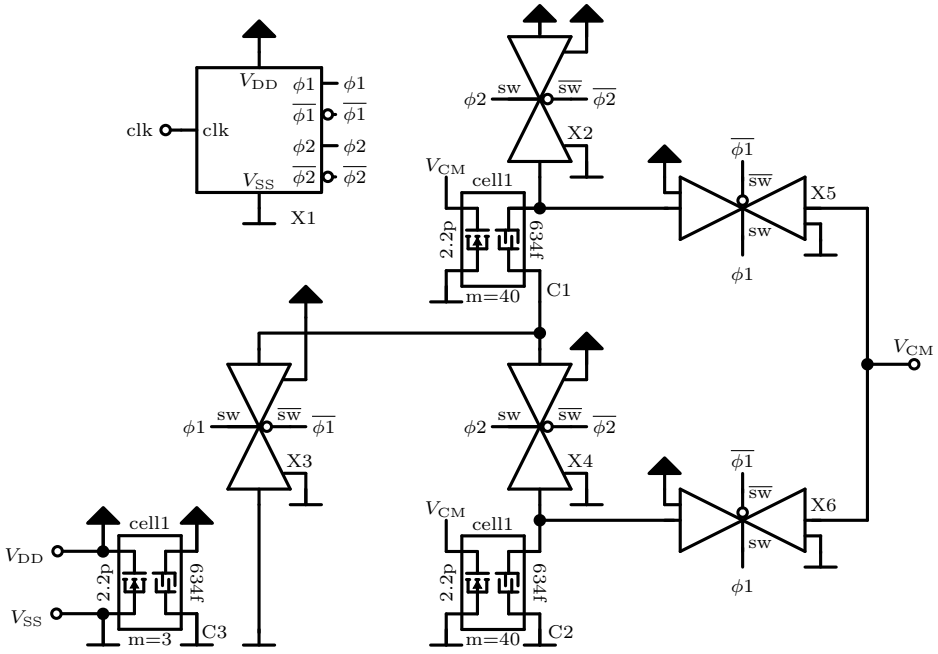



**Figure 2.23:** Schematic of the $V_{CM}$ generator. X1 is a non-overlapping-clock generator and X2-6 are bidirectional gates. C1 and C2 are the decoupling cells `adc_noise_decoup_cell1`. $\phi1$ and $\phi2$ are the two complementary switching phases of the switched-capacitor voltage divider.

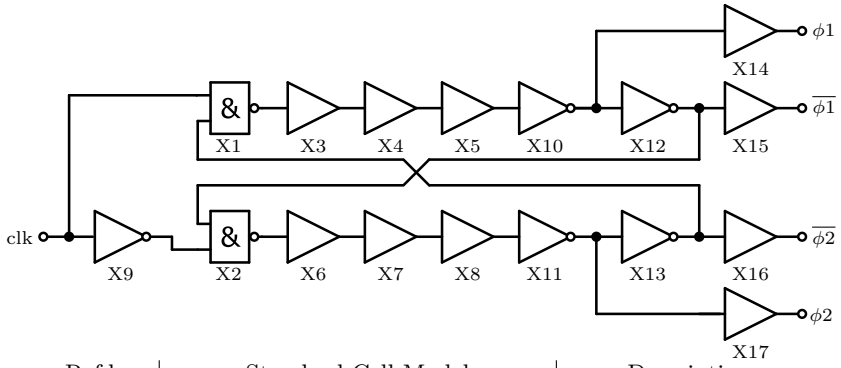
## 2.9 Switched-Capacitor Voltage Divider

In the sampling phase, the common-mode voltage $V_{CM}$ is connected to every DAC-matrix capacitor bottom plate. For the generation of this reference voltage $V_{CM} = V_{DD}/2 = 0.9\,V$ a switched-capacitor voltage divider [9] is used. The $V_{CM}$ generator is assembled as shown in Fig. 2.23 using transmission gates, previously designed capacitors `adc_noise_decoup_cell1`, and a non-overlapping clock generator built entirely with SKY130 digital standard cells.

The switched-capacitor voltage generator schematic is shown in Fig. 2.23. In phase $\phi2$ the MIM capacitors of C1 and C2 connect in series, they are charged between $V_{DD}$ and $V_{SS}$. Each MIM capacitor is therefore charged to $V_{DD}/2 = 0.9\,V$ through a capacitive voltage divider. In phase $\phi1$, the MIM capacitor structures are being switched in parallel to the MOS capacitor structures, and charge balances between the parallel capacitors. By repeating this sequence the voltage on $V_{CM}$ converges to $V_{DD}/2 = 0.9\,V$.

### 2.9.1 Non-Overlapping-Clock Generator

For the $V_{CM}$ regulation, a differential two-phased clock signal $\phi1$ and $\phi2$ without overlapping the active states [11] is needed. The clock generator schematic as seen in




| Refdes. | Standard Cell Model | Description |
|---|---|---|
| X1-2 | `sky130_fd_sc_hd__nand2` | 2 input NAND |
| X3-8 | `sky130_fd_sc_hd__dlymetal6s6s` | 6 inverter delay-chain |
| X9-13 | `sky130_fd_sc_hd__inv1` | Inverter, size 1 |
| X14-17 | `sky130_fd_sc_hd__buf_4` | Buffer, size 4 |

**Figure 2.24:** Schematic of the non-overlapping clock [11]. The clock generator generates two-phased differential clock signals without overlapping.

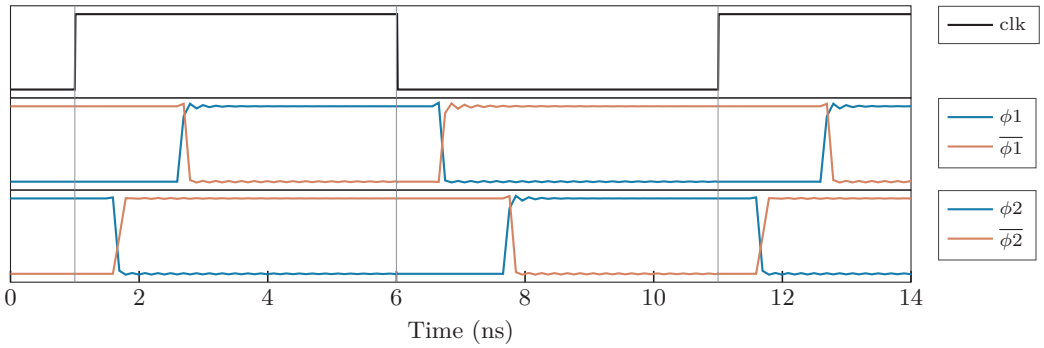



**Figure 2.25:** Transient-simulation waveforms of the non-overlapping-clock generator. $\phi1$ becomes active when `clk` is `HIGH`, $\phi2$ at `clk`=`LOW`. There is no moment where $\phi1$ and $\phi2$ active states overlap.

Fig. 2.24 has been implemented in `xschem` (`adc_vcm_clkgen.sch`) using only SKY130 digital standard cells of the high-density library `sky130_fd_sc_hd`[5].

The circuit has been simulated with `ngspice` using the analog models of the digital standard cells in a transient sweep. The simulation result plotted in Fig. 2.25 shows that the circuit is generating differential and non-overlapping two-phased signals.

### 2.9.2 Implementation

In the last subsection, the circuit behind the non-overlapping-clock component X1 in Fig. 2.23 has been defined. For the capacitors (C1, C2, C3) the previously designed `adc_noise_decoup_cell1` can be used. For C1 and C2, 40 parallel cells per capacitor have a total capacitance of 25.4 pF MIM-capacitance and 90 pF MOS-capacitance each, the voltage $V_{\mathrm{CM}}$ is therefore buffered with $C_{\mathrm{VCM}} = 2 \cdot 90\,\mathrm{pF} = 180\,\mathrm{pF}$. 3 additional capacitor cells decouple the source voltage $V_{\mathrm{DD}}$ from noise. The complementary

[5]Further information about the high-density standard cells can be found in the Skywater documentation `https://antmicro-skywater-pdk-docs.readthedocs.io/en/test-submodules-in-rtd/contents/libraries/sky130_fd_sc_hd/README.html`.

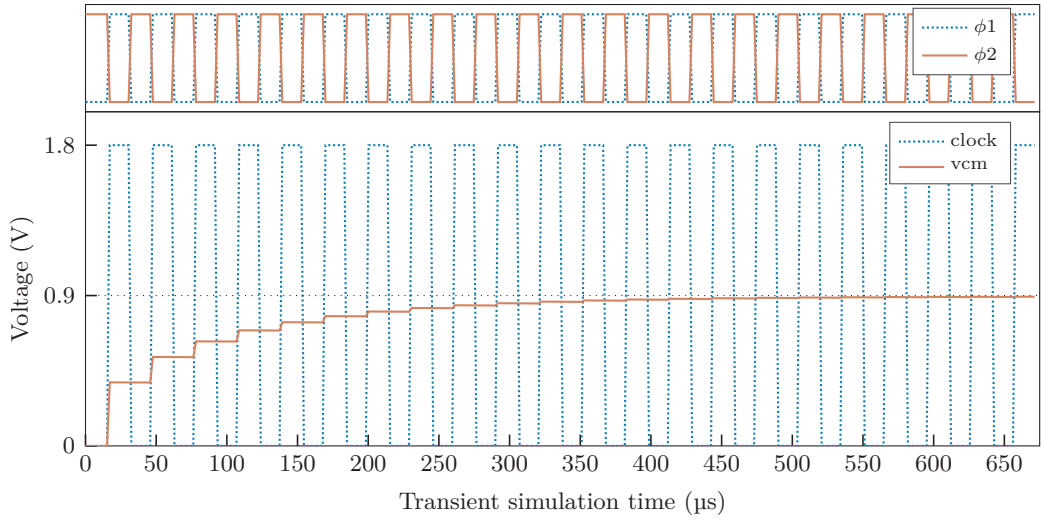
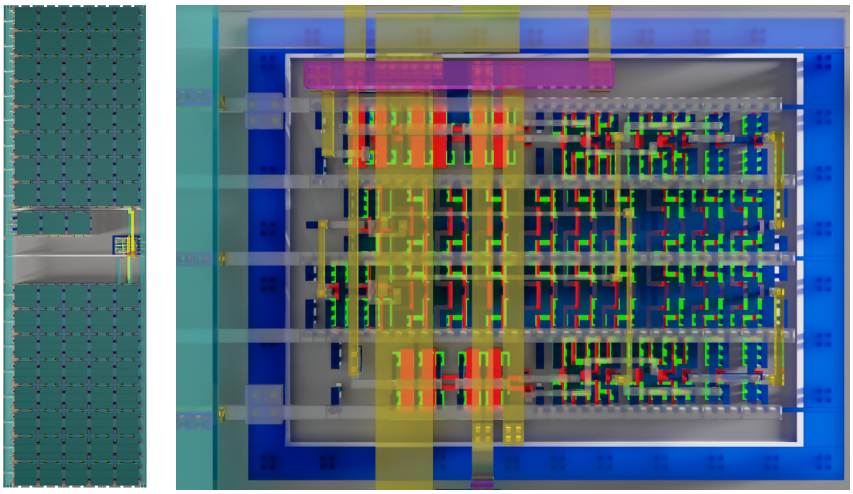



**Figure 2.26:** Transient simulation results of the switched-capacitor voltage divider. The divider has been sourced with a $f = 32\,768\,\text{Hz}$ clock. Shown on top are the two phases of the non-overlapping-clock generator $\phi 1$ and $\phi 2$. On the bottom are the source-clock signal and the output voltage $V_{\text{CM}}$ leveling at 0.9 V.

switches X2-X6 have different requirements than the S & H switch in the DAC-matrix. With an expected clock frequency of $f_{\text{clk,vcm}} = 32\,768\,\text{Hz}$ [9] the switch open- and close-phases are longer, also the capacitive load is higher, whereas the switch has been designed with eased requirements. The chosen MOSFET dimensions are $W = 1\,\mu\text{m}$ and $L = 0.5\,\mu\text{m}$ for PMOS and NMOS. Compared to the previously designed S & H switch, a smaller width has been chosen since the timespan for charging the capacitors is longer, no charge compensation MOSFETs have been added since the capacitive load is huge, and an average gate length has been chosen for reduced leakage current.



(a) (b)

**Figure 2.27:** The layout of the $V_{\mathrm{CM}}$ generator `adc_vcm_generator.gds`. (a) Rendering of the macro, capacitor cells on top and bottom occupy the majority of the area. Vertical power rails $V_{\mathrm{DD}}$ and $V_{\mathrm{SS}}$ are located left and right on `metal4`, also a power rail for $V_{\mathrm{CM}}$ has been added. The macro is designed with intentional empty space in the middle, this allows the routing of signals through the layout. (b) Zoomed image showing the digital standard cell area in the standard cell power-distribution network including the non-overlapping-clock generator and the bidirectional switches.

# Chapter 3

# "Digital-Friendly" Analog Design

For the asynchronous self-clocking mechanism, the clock generation implementation from [11] has been redesigned allowing the usage of a digital workflow for GDSII generation. The reason behind this design choice is a steady trend towards "digital-friendly" analog circuits in a highly-automated design flow [19]. The SKY130 PDK provides digital standard cells (inverter, NOR, ...) with digital standard cell libraries [7]. The challenge is to implement synthesizable signal delays using Verilog hardware description language (Verilog HDL).

This chapter begins with the design of a SKY130 delay custom standard cell `sky130_mm_sc_hd_dlyPoly5ns` using its signal transition time to implement delays in a digital circuit. A miniaturized implementation of an analog delay circuit [11] has been embedded in the layout using a SKY130 standard cell geometry. The digital workflow is capable of placing and routing the custom standard cell in the digital standard cell grid.

This is followed by an analysis of the transition delays in available SKY130 standard cells. The edge detection and clock generation circuits are explained, and the first revision of the self-clocking loop is presented with available standard cells. It is proof of work, but it shows the inefficiency of using only the available clock delay standard cells to apply delays. The final layout has been hardened with `OpenLane` using the new custom standard cell `sky130_mm_sc_hd_dlyPoly5ns`, which has been designed in this work. The development tools used in this workflow are the same as the used tools for the analog designs, as shown in Sec. 2.

## 3.1 Delay Custom Standard Cell

This chapter covers the design of a delay circuit embedded in the SKY130 single-row standard cell geometry using the layout tool `magic`. The delay cell can be instantiated in a digital workflow after the `.gds`, `.lef` and `.lib` files of the custom standard cell have been generated. The `.gds` file contains the detailed layout of the cell, whereas the `.lef` file describes an abstracted view containing the pin positions and obstructions (this eases the routing and placing process). The Liberty file (`.lib`) simplifies the

I/O timing and power calculations of the abstracted cell for timing analysis without the need for extensive analog simulations. The final layout `.gds` of the delay custom standard cell can be seen in Fig. 3.3.

### 3.1.1 Delay Circuit

The analog delay component from the clock generation loop in [11] is intended to maximize the digital signal transition time through the cell. The circuit consists of a weak inverter as input, binary-coded capacitors with enable circuit as variable capacitive load, and an inverting Schmitt trigger as the output stage. The total delay in the referenced solution can be changed by enabling or disabling the binary fractioned capacitors. Unfortunately, any changes to the delays need to be done in the analog domain. To bypass this limitation, a miniaturized version of the delay cell circuit shown in Fig. 3.1 with a fixed capacitance has been designed. It has a signal transition time of approximately 5 ns. The layout of this miniaturized cell is still done in the analog domain, but the delay cells with 5 ns each can be cascaded. The total delay is configured by instantiating delay cells in series, which allows the implementation of changes in the design using Verilog HDL. The designed delay cell is using the form factor of a SKY130 digital standard cell. As a result, it allows the instantiation in the gate-level description of a design and therefore an analog, time-dependent layout can be generated using a digital workflow.

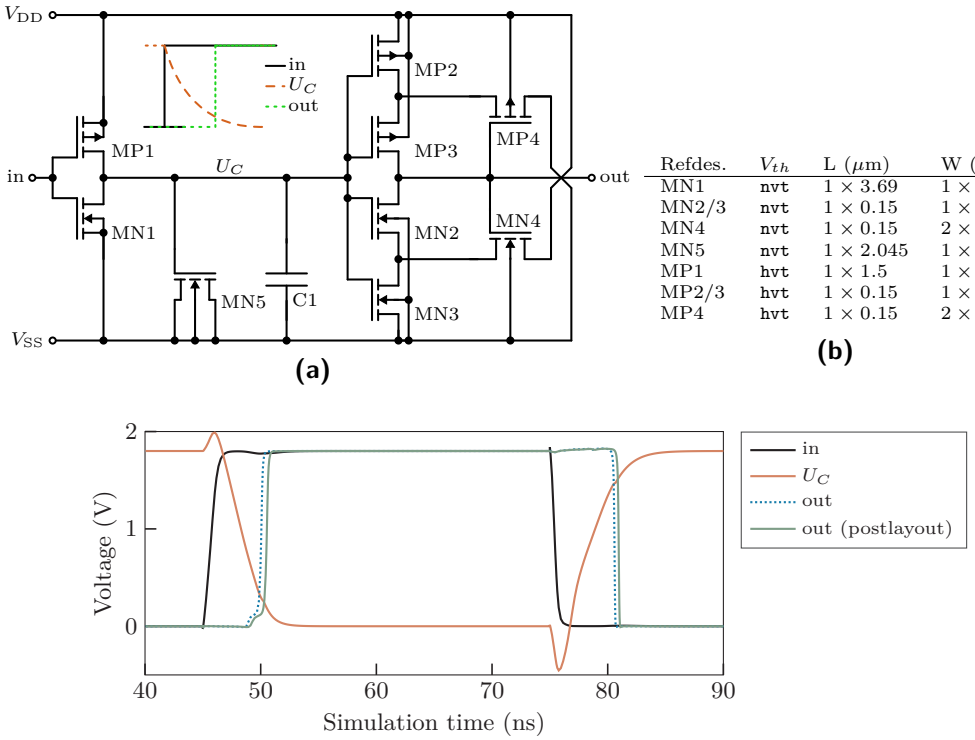
### 3.1.2 Standard Cell Geometry

The standard cell geometry depends on the used standard cell library, for the high-density library `_hd_` the single-height standard cell abutment box (`FIXED_BBOX`) must align to a grid with $N \times 0.46\,\mu$m length and $8 \times 0.34\,\mu$m height [20] where N is a natural number that can be chosen. The power rail and via dimensions of abutting standard cells are also predefined, they must be able to overlap without generating DRC errors. The dimensions of `sky130_mm_sc_hd_dlyPoly5ns` can be seen in Fig. 3.2. The substrate connections must not be added to the standard cell layout since `_hd_` is a tap-less library, they can be added though, but usually dedicated tap cells[1] are placed in the standard cell grid. If the custom standard cell is designed without tap cells then the power and MOSFET bulk connections are implemented as ports in the schematic and also in the Verilog files. In the SKY130 PDK, these ports are usually named as `VPWR` for the power, `VPB` for the PMOS bulk, `VGND` for the ground and `VNB` for the NMOS bulk connections.

### 3.1.3 Layout in `magic`

The custom standard cell must be able to abut any standard cell in all possible directions and orientations without violating the DRC rules. It's advantageous to use existing standard cells as a template, which can be observed in the technology library

[1] For a tap cell connecting to both `VPWR` and `VGND` see `https://antmicro-skywater-pdk-docs.readthedocs.io/en/test-submodules-in-rtd/contents/libraries/sky130_fd_sc_hd/cells/tapvpwrvgnd/README.html`.




| Refdes. | $V_{th}$ | L ($\mu$m) | W ($\mu$m) |
|---|---|---|---|
| MN1 | nvt | $1 \times 3.69$ | $1 \times 0.42$ |
| MN2/3 | nvt | $1 \times 0.15$ | $1 \times 0.42$ |
| MN4 | nvt | $1 \times 0.15$ | $2 \times 0.42$ |
| MN5 | nvt | $1 \times 2.045$ | $1 \times 1.375$ |
| MP1 | hvt | $1 \times 1.5$ | $1 \times 0.80$ |
| MP2/3 | hvt | $1 \times 0.15$ | $1 \times 0.80$ |
| MP4 | hvt | $1 \times 0.15$ | $2 \times 0.80$ |

**Figure 3.1:** (a) Circuit of the analog delay element. The input stage is an inverter with low W/L. The load of the input stage is the MOS-capacitor MN5 parallel to the parasitic capacitance C1 = 2.1 fF using metal layers up to `metal1`. The output stage is an inverting Schmitt trigger. (b) Table specifying MOSFET types and dimensions. Type `nvt` refers to the nominal $V_{th}$ doping, `hvt` is high $V_{th}$. (c) Simulation results of the delay cell, rise transition delay is $T_{\text{rise}} = 4.62\,\text{ns}$, fall transition delay is $T_{\text{fall}} = 5.48\,\text{ns}$.

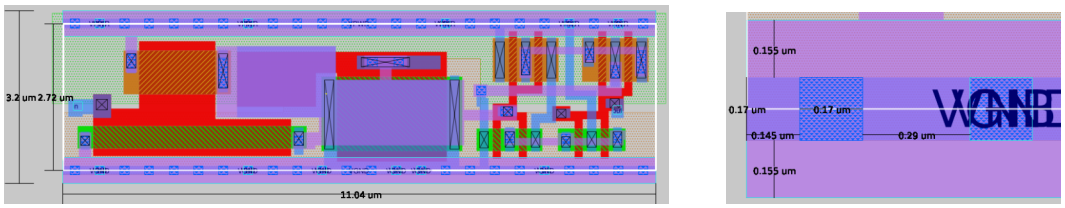



**Figure 3.2:** High-density standard cell geometry from cell `sky130_mm_sc_hd_dlyPoly5ns`. (a) Abutment box with $24 \times 0.46\,\mu\text{m} = 11.04\,\mu\text{m}$ length and $8 \times 0.34\,\mu\text{m} = 2.72\,\mu\text{m}$ height. (b) Dimensions of the horizontal power rail on layer `m1` with vias to `li`. Via size is $0.17\,\mu\text{m} \times 0.17\,\mu\text{m}$, `li` width is $0.17\,\mu\text{m}$, `m1` rail width is $0.48\,\mu\text{m}$.

at `libs.ref/sky130_fd_sc_hd/gds`. It has been observed that the design rules are easier to follow if the transistor types match the ones used in the standard cell library. Using the high-density library, those are `hvt` PMOS and `nvt` NMOS.

The custom standard cell needs a tag to enable the DRC rules specifically for standard cells. The hidden standard cell technology layers must be unlocked in `magic` using command `tech unlock *` before they can be used. Then, the NMOS and PMOS layers can be replaced using the layers `scnmos` and `scpmoshvt`. As a result, `magic`

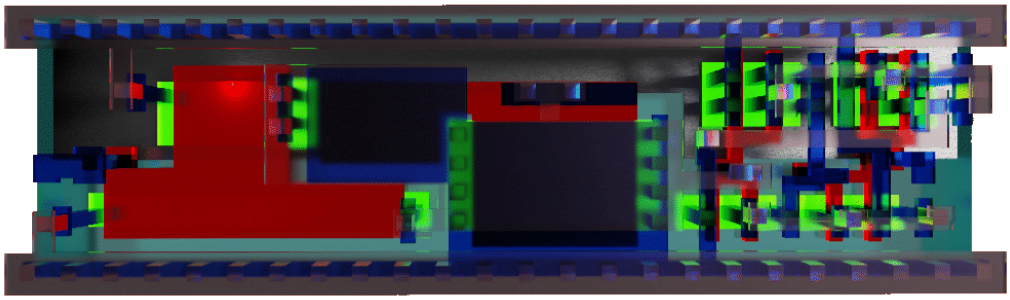


**Figure 3.3:** Delay cell layout with the standard cell geometry for SKY130 high-density designs. Diffusion areas are colored green, polysilicon is red, local interconnect is blue, and layer `m1` is transparent. From left to right: Input inverter stage, MOS- and MIM-capacitor load, inverting Schmitt Trigger output stage.

will recognize the standard cell MOSFETs and adds the standard cell identification layer (areaid.sc) when generating the `.gds` file using the `FIXED_BBOX` property.

For the generation of the abstract view of the cell, the `.lef`[2] file, some properties need to be set in `magic` beforehand to allow the digital toolchain to handle the cell properly. The command `property LEFclass CORE` tags the extracted cell as a standard cell, which is intended to be placed in a standard cell grid. The command `property FIXED_BBOX {AX AY BX BY}` sets the standard cell abutment boundary box and `property LEFsite unithd` specifies to use the unit-distance from a high-density standard cell grid. Furthermore, the directions and types of the signal in-/outputs and power ports need to be defined to be correctly handled by the router and power distribution network generator. The executed `tcl` commands for the configuration of the ports are summarized in A.2 Listing A.3.

### 3.1.4 Liberty File Generation

The `OpenLane` workflow expects a Liberty file `.lib` with timing and power models for the custom standard cell. The contained data is used to calculate the path delays to check if timing constraints in the design are met. The hardened macro using the custom standard cell is going to be an asynchronous gate-level design, whereas no external clock is present and the static timing analysis (STA) and optimizations are not intended to be used for the designs in this chapter. This allows the undocumented usage of a dummy Liberty file.

The dummy file `sky130_mm_sc_hd_dlyPoly5ns.lib` has been generated by copying the Liberty tables of an existing buffer while adapting the cell name, area and ports of the custom standard cell. The data in the tables (power and timing) have not been changed, although they do not model the real behavior of the custom standard cell[3]. The tools used in the `OpenLane` flow assume the data is valid and, as a result, complete the RTL2GDSII flow. Note that it is indispensable to verify the function

[2] See the LEFDEFREF reference manual for further information about the LEF file format.

[3] The correct values need to be determined by simulation or by measuring test structures on silicon.

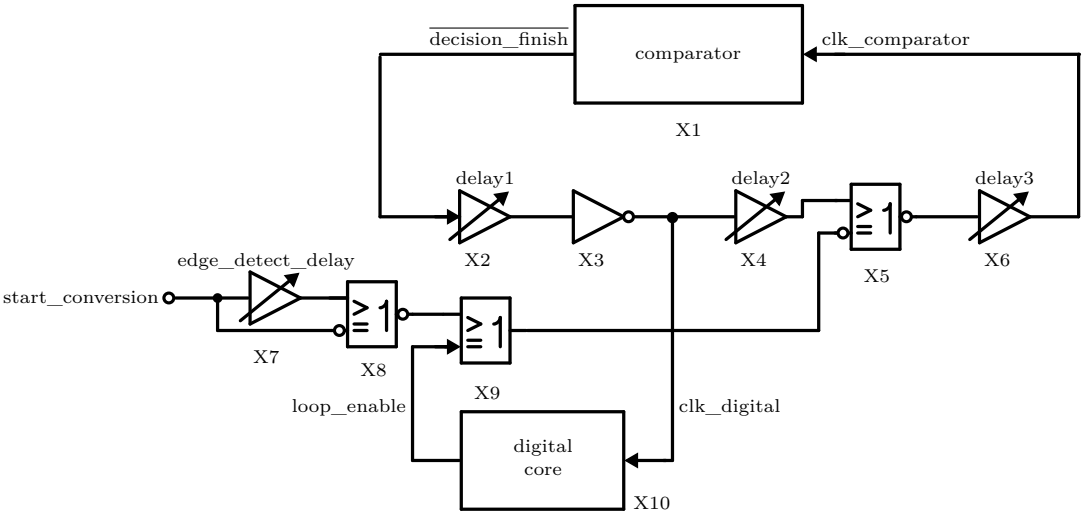



**Figure 3.4:** Block diagram of the self-clocking mechanism in the ADC [11].

of the hardened macro in analog post-layout simulation. If a valid Liberty file is necessary, then it is possible to search for command `write_timing_model` in the `OpenRoad` documentation, but the exploration of this feature was not part of this work.

## 3.2 Clock Generator with Edge Detection

The ADC needs two clock sources, one slow clock for the switched-capacitor voltage divider (design-frequency for a biosignal sensor application $f = 32.768\,\text{kHz}$ [11]) and a fast clock signal `clk_digital` to control the SAR-ADC logic and the comparator. The faster clock is asynchronously generated on-chip as seen in Fig. 3.4 and is active only while the ADC is converting. This saves energy since no power is wasted by a permanently running clock [11]. For the clock generator loop, a delay module is needed. This delay can be generated using the signal transition times of standard cells. The implementation of this module has been distilled to 3 possible solutions allowing the usage of digital standard cell placement and routing:

1. Instantiate a chain of delay cells from the standard cell library.
2. Instantiate a chain of small buffers/inverters followed by big buffers/inverters as load.
3. Definition of a custom standard cell with delay and instantiate it in Verilog HDL.

Since the first and the second possible solution are area-inefficient, the third solution was implemented using a new custom standard cell `sky130_mm_sc_hd_dlyPoly5ns`.

### 3.2.1 Self-Clocking Loop and Edge Detection

The self-clocking mechanism shown in Fig. 3.4 is enabled or disabled by either releasing or blocking the ring oscillator loop. When the comparator input `clk_comparator` is

LOW=0 V then the comparator is in reset mode and the output $\overline{\texttt{decision\_finish}}$ is set to HIGH=1.8 V. This signal is delayed by component X2 by signal transition time $T_{\text{delay1}}$. Afterward, inverter X3 resets the generated digital clock `clk_digital` to LOW. Signal `clk_digital` is then further delayed by X4 with $T_{\text{delay2}}$ until it reaches the loop-enable component X5. If the inverted input of X5 is LOW, then the NOR output is also LOW and `clk_comparator` keeps its previous state. As a result, the loop is in a stable condition.

A positive edge at the trigger signal `start_conversion` releases the loop. In this case, the output of X5 becomes HIGH. This signal is once again delayed through $T_{\text{delay3}}$, but now, the node `clk_comparator` transits to HIGH and triggers the comparator to do a comparison. A finished comparison is signalized when $\overline{\texttt{decision\_finish}}$ changes to LOW. After $T_{\text{delay1}}$ the digital core is triggered because signal `clk_digital` changes to HIGH, and the state machine sets signal `loop_enable` HIGH to release the loop cycle until a full conversion cycle has been completed. Once again X6 is passed, `clk_comparator` changes back to LOW and the first loop cycle is completed. The loop generates a clock signal with a period calculated as shown in Eq. 3.1, it is twice the transition time through 3 delay cells and the comparator conversion delay. When the total number of cycles for one analog-to-digital conversion has been reached, then the clock generation is halted again by disabling signal `loop_enable` until `start_conversion` releases the loop again.

$$T_{\text{clk}} = 2T_{\text{delay1}} + 2T_{\text{delay2}} + 2T_{\text{delay3}} + T_{\text{conversion}} \tag{3.1}$$

The components X7 and X8 build an edge-detect circuit to suppress multiple conversions from a single positive signal edge of the trigger signal $\overline{\texttt{start\_conversion}}$. The high-period of the transitioned signal is limited to the maximum timespan of $T_{\text{edge\_detect\_delay}}$. This ensures only one conversion cycle is triggered if the input signal $\overline{\texttt{start\_conversion}}$ is not reset.

#### 3.2.1.1 `OpenLane` RTL2GDSII Configuration

`OpenLane` is an automated RTL to GDSII flow and is based on several components using `OpenRoad`, `Yosys`, `magic`, `netgen` and custom scripts [21]. In a regular operation with standard configurations, the workflow in Fig. 3.5 is executed and generates a `.gds` layout from the sourced register transfer level (RTL) Verilog HDL files.

For the goals in this chapter, synthesis and the optimizations in the standard workflow from `OpenLane` would alter the gate-level (GL) design and therefore need to be skipped. Synthesis can remove logic if optimizable logic is identified, as inverters in series e.g. `assign a = not(b)` and `assign b = not(c)` (delay chain with inverters) can result in the removal of both inversions if `b` has no direct path to any output. The solution is to explicitly instantiate the standard cells as black box instances, instead of an implicit logical function in RTL. Configuration `SYNTH_READ_BLACKBOX_LIB` in `config.tcl` will allow the standard cell instances to be interpreted as black boxes, therefore no

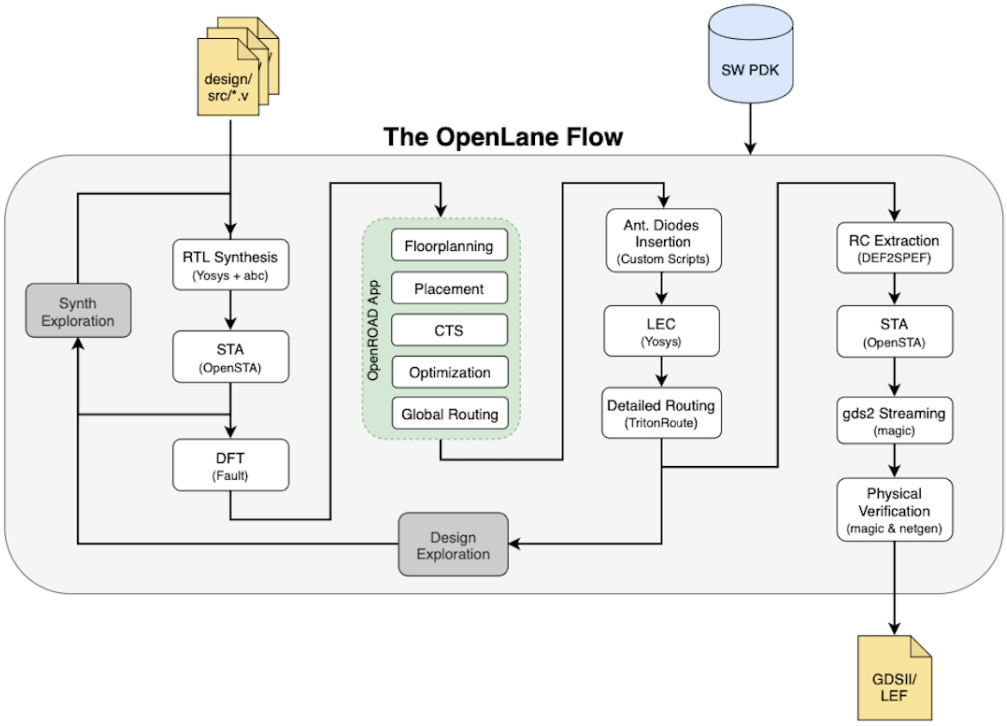



**Figure 3.5:** The `OpenLane` flow [21].

error message will be thrown if instances do not have a synthesizable RTL function. To further ensure that the synthesizer is not messing with the intended cell structure, the Verilog HDL can be completely described on the gate-level instead of mixing RTL with GL logic.

**Listing 3.1:** Example for gate-level instantiation of SKY130 standard cells using Verilog HDL.

```
1  sky130_fd_sc_hd__inv_2 inv1 (.A(b),.Y(c));
2  sky130_fd_sc_hd__inv_2 inv2 (.A(a),.Y(b));
```

Another limitation is the use of optimizations meant for digital circuits executed in the placement and global routing stage shown in Fig. 3.5. The optimizer might interpret the signal transition time through delay cells (buffers) as too high/low and shrink their size for better timing or area, these optimizations can be turned off in the `OpenLane` configuration file `config.tcl`. The respective variables for the placer are `PL_RESIZER_DESIGN_OPTIMIZATIONS` and `PL_RESIZER_TIMING_OPTIMIZATIONS`, the global router optimizations can be turned off with `GLB_RESIZER_TIMING_OPTIMIZATIONS`. In this work it has not been determined which optimizers can be left on, instead, the Verilog HDL files have been designed using only gate-level description with an inherently good structure to allow the disabling of all optimizations.

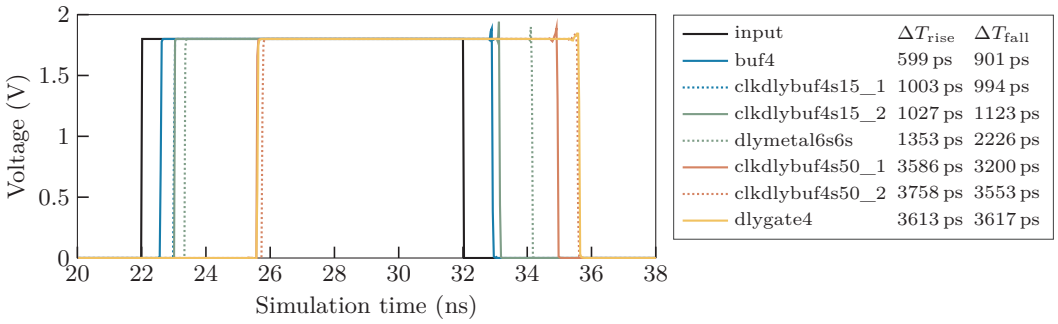



**Figure 3.6:** Simulated signal rise $\Delta T_{\text{rise}}$ and fall $\Delta T_{\text{fall}}$ transition delays through digital standard cells (10 in series each) from the SKY130 high-density standard cell library `sky130_fd_sc_hd`.

### 3.2.2 Solution with SKY130 Standard Cells

This section describes the solution using `clkdlybuf4s50_2` digital standard cells for the delay components in the clock loop. The solution with these cells has not been chosen for the final layout, because the high number of standard cells is reducing area efficiency.

#### 3.2.2.1 SKY130 standard cell delays

For a solution with delay modules made from available standard cells (buffers and clock delay cells), the signal transition times of the individual cells need to be known beforehand. A testbench has been set up using `xschem` comparing the signal transition delays of 7 selected standard cell types. The results shown in Fig. 3.6 have been obtained with 10 standard cells of each type in a series configuration. They show that `clkdlybuf4s50_2` and `dlygate4` have the highest transition times of the compared cells with 3.6 ns, whereas `clkdlybuf4s50_2` was chosen for the clock loop solution.

#### 3.2.2.2 HDL Implementation

The delay module is generated by connecting a parametrizable number `N_TIMES` of clock delay buffers `clkdlybuf4s50_2` in series. A single cell `clkdlybuf4s50_2` reached a signal transition delay of $T_{\text{delay}} = 0.36\,\text{ns}$ in the analog simulation. To be able to delay the input signal by $T_{\text{delay}} = 100\,\text{ns}$, a total number of `N_TIMES` $= 270$ cells is necessary. The respective Verilog HDL for one delay chain is shown in Listing 3.2.

**Listing 3.2:** Delay chain HDL generating `N_TIMES` clock delay buffers in a row for the clock generation loop using only standard cells.

```
1 generate
2    for(j=0;j<N_TIMES;j=j+1) begin
3    sky130_fd_sc_hd__clkdlybuf4s50_2 delay_unit (.in(signal_w[j]),
     ↪  .out(signal_w[j+1]));
4    end
5 endgenerate
```

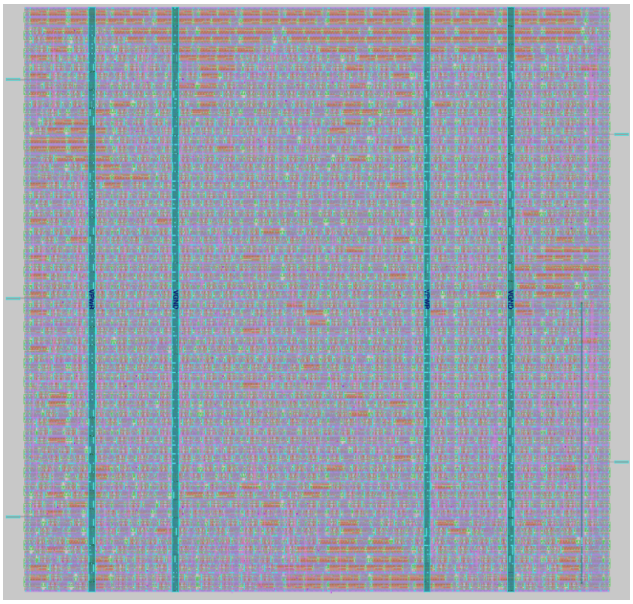


**Figure 3.7:** Hardened macro of the clock generation circuit with edge detection using `clkdlybuf4s50_2` standard cells as the delay components, generated with the `OpenLane` RTL2GDSII flow.

The delay components of the clock loop generator shown in Fig. 3.4 determine the sample speed of the SAR-ADC. The delays have been specified to add signal transition times that are comparable to the implemented delays in a SAR-ADC used for bio-sensing [11]. The signal transition delays $T_{\text{delay1}}$, $T_{\text{delay2}}$ and $T_{\text{delay3}}$ were set to approximately $T_{\text{delay}} = 100\,\text{ns}$. The target for the signal transition time of component $T_{\text{edge\_detect\_delay}}$ has been set to 200 ns. As a result, the Verilog HDL modules `adc_edge_detect_circuit` and `adc_clk_generation` have been implemented, their modules are shown in Listing A.1. Both modules were combined in the Verilog gate-level HDL `adc_clkgen_with_edgedetect_native`. 1350 instances of standard cell `clkdlybuf4s50_2` were necessary to build the layout with the specified signal transition delays.

#### 3.2.2.3 Hardening with `OpenLane`

The design `adc_clkgen_with_edgedetect_native` was hardened using the automated `OpenLane` RTL2GDSII flow. The hardening process was initiated by executing `flow.tcl -design adc_clkgen_with_edgedetect_native` on the command line, the used configuration file `config.tcl` is shown in Listing A.2. The result in Fig. 3.7 shows a clock generator macrocell with dimensions of $185.5\,\mu\text{m} \times 175.0\,\mu\text{m}$. Because of the huge size, this solution has been discarded. Instead, a different approach using a custom standard cell has been implemented to increase area efficiency.

### 3.2.3 Solution with Custom Standard Cell

The self-clocking loop with edge detection logic as shown in Fig. 3.4 has been implemented in Verilog HDL using the previously designed custom standard cell `sky130_mm_sc_hd_dlyPoly5ns` as delay element. The resulting macro allows the

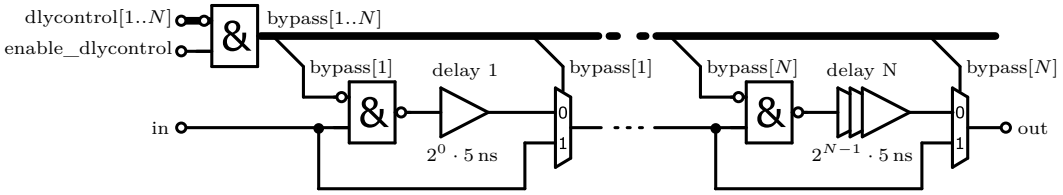



**Figure 3.8:** Structure of the parametrizable delay module using binary delay sequences.

configuration of the delay components X2, X4 and X6 from 0 ns to 155 ns in 5 ns steps with 5 control wires each. Furthermore, the edge detection delay X7 can be set in the range 0 ns to 310 ns with 6 control wires. The dimensions of the hardened macro are only 60 $\mu$m × 171 $\mu$m which is an area reduction of 68% compared to the solution with `clkdlybuf4s50_2`, although the logic for parametrizable delays was not included and the delay range has additionally been extended by 60%.

#### 3.2.3.1 Delay Module

The delay modules need to be parametrizable to allow the configuration of the digital clock frequency and the edge detection pulse width. The block diagram of the proposed delay module is shown in Fig. 3.8. Organizing the delay chains in $N$ binary steps ($2^0 \cdot 5\,\text{ns}$, $2^1 \cdot 5\,\text{ns}$, ... , $2^{N-1} \cdot 5\,\text{ns}$) allows hardware-efficient configuration using $N$ configuration wires. NAND-gates before the delay inputs have been implemented to disable unused delay cells, this reduces the power consumption of bypassed delay cells.

#### 3.2.3.2 Implementation

The HDL implementation of module `adc_clkgen_with_edgedetect` can be seen in Listing A.4. Additionally to the clock generation and edge detection modules, a delay stage for the DAC sample signal is integrated. The sample signal is delayed through a stage of four delay cells `dlymetal6s6s_1` and buffered at the output with cell `buf_4`. The reason for this addition is to delay the sample signals precautiously because the S & H switch gates must isolate the DAC capacitor from the analog input signal before the DAC capacitor bottom plates change their value, the DAC top cap voltages can be falsified otherwise.

The final layout `adc_clkgen_with_edgedetect.gds` as shown in Fig. 3.9 has been hardened using `OpenLane` with the configurations shown in Listing A.5 in Sec. A.3. `OpenLane` has been configured to generate the power distribution network (PDN) on layers `m2` and `m3` to keep the above layers free for top-level PDN generation. The core hardening configuration has been set to connect the standard cell power grid on `m1` to vertical power rails `m2` with horizontal `m3` power rails above. As a result, a top-level power distribution network instantiating the clock generator macro can connect vertical `m4` rails to the macro PDN on `m3`.

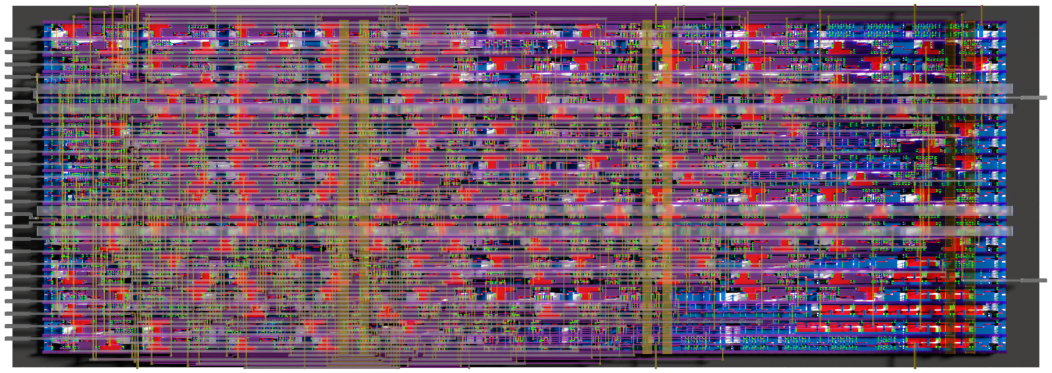
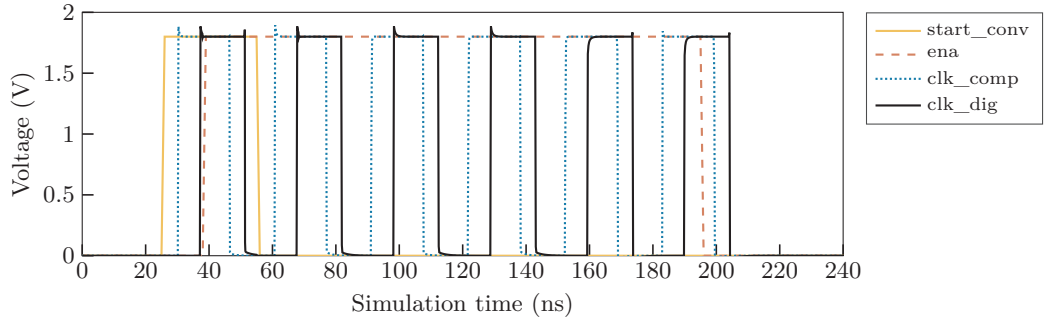
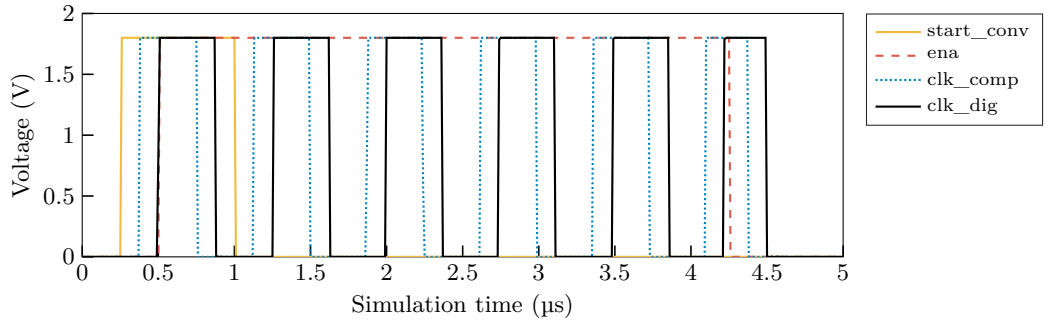


**Figure 3.9:** Macrocell `adc_clkgen_with_edgedetect.gds` implementing the asynchronous clock generation for digital- and comparator-clock, trigger signal edge detection, and sample signal delaying. The macro requires an area of 60 $\mu$m × 171 $\mu$m and uses layers up to `m3`.




**(a)**




**(b)**

**Figure 3.10:** ADC clock generator post-layout simulation. The clock generation starts with the assertion of signal `start_conv`, `ena` keeps the loop unblocked. When signal `ena` is de-asserted then the clock generation stops. (a) Fast clock with configuration `dlycontrol_1/2/3=00001` and `dlycontrol_4=000011`, $f_{\mathrm{clk\_dig}} = 32.8\,\mathrm{MHz}$. (b) Slow clock with setting `dlycontrol_1/2/3=11111` and `dlycontrol_4=111111`, $f_{\mathrm{clk\_dig}} = 1.35\,\mathrm{MHz}$.

# Chapter 4

# Digital Design

This chapter covers the design of the digital logic in the SAR-ADC. The digital part handles the state machine, configurations, and the processing of the SAR algorithm. First, the digital workflow is described, and a method to simulate digital logic with analog macros is presented. Then, the structure of the digital core module is specified. The digital core module `adc_core_digital.v` includes the DAC matrix row and column decoders `adc_row_col_decoder.v`, the synchronous digital core logic `adc_core_digital.v`, and the asynchronous oversampling logic `adc_osr.v`. Moreover, these sub-modules are described in detail. The row and column decoders translate the current DAC code to the matrix row and column wire control signals, and the module allows the selection of a sequential or symmetrical scan mode. The digital core logic handles the non-binary SAR algorithm, averaging of the LSB bits, clock loop control, and the calculation of the 12 bit result with averaging of the 4 least significant SAR weights. The oversampling module, which is asynchronously clocked using a strobe signal, calculates the 16 bit oversampled result.

## 4.1 Digital Workflow

The digital workflow begins with the specification of the behavior and the interface of the digital core logic. A divide-and-conquer strategy separates the functions of the core module into smaller sub-modules. This reduces the design complexity as a result of isolating separated functions into sub-modules. The result of this approach is reflected in the module hierarchy. The low-level modules are designed and simulated as standalone modules. Afterward, the low-level modules are combined and simulated in the core module. When the core module is finalized, it can be exported to a behavioral description in the XSPICE format. This allows low-complexity mixed-mode simulation using SKY130 technology before the top-level layout has been generated.

The following list summarizes the current (03/2023) sources used for the digital design toolset in this work:

- SKY130: Open-PDKs installer (`github.com/RTimothyEdwards/open_pdks`).

- xschem: Electrical circuit simulator with internal waveform viewer and mixed-signal support (`github.com/StefanSchippers/xschem`).
- ngspice: Mixed-signal simulator (`github.com/ngspice/ngspice`).
- gaw3: Waveform viewer (`github.com/StefanSchippers/xschem-gaw`).
- Verilator: Verilog simulator (`github.com/verilator/verilator`).
- iverilog: Icarus Verilog compiler (`github.com/steveicarus/iverilog`).
- gtkwave: Waveform viewer (`github.com/gtkwave/gtkwave`).
- qflow: Digital end-to-end synthesis flow for ASIC designs (`github.com/RTimothyEdwards/qflow`).

### 4.1.1 Design and Simulation

The digital logic is implemented using Verilog hardware description language (Verilog HDL) at the register-transfer level (RTL) with technology-independent syntax. Syntax linting of the HDL is checked using `Verilator`, and compilation of the test benches is done with `iverilog`. The tool `vvp` executes the compiled logic and writes the simulation results into a file, and the waveform-viewer `gtkwave` plots the waveforms obtained from the simulation.

```
iverilog -g2005 -I "../rtl/" -s design_tb -o dump.vvp design_tb.v
verilator --lint-only -Wall ../rtl/design.v
vvp dump.vvp
gtkwave dump.vcd
```

### 4.1.2 Mixed-Signal Simulation

The digital standard cells are made of analog NMOS and PMOS MOSFETs in a CMOS configuration. The analog post-layout simulation of the top-level layout in Sec. 5 results in a `.spice` netlist containing more than 28.000 MOSFETs. This leads to a high simulation complexity, and it is favorable to reduce simulation complexity for pre-layout validation.

The XSPICE extension of `ngspice` provides an interface between digital behavioral models and analog `.spice` simulations. The interfaces between the analog and the digital domain are specified as the subcircuits `toana_1v8` and `todig_1v8`. The digital standard cell subcircuits must be converted to a behavioral model, which is defined by parameters such as the rise delay, fall delay, transition time, offset and others. This allows the pre-layout simulation of a mixed-signal circuit which is processed several orders of magnitude faster than the analog simulation.

The conversion of an RTL model to the `.xschem` netlist format has been documented in detail on the GitHub repository of the SAR-ADC[1]. The documentation covers

[1] `github.com/iic-jku/SKY130_SAR-ADC1/blob/main/doc/mixed_signal_simulation.md`.

multiple options to generate the digital behavioral model. In this work, only the most regularly used workflow for the simulation of this SAR-ADC is presented. First, the Verilog RTL description of the digital module must be synthesized to a SKY130 gate-level description. This can be done by setting up a minimal `OpenLane`[2] project, followed by running the `OpenLane` flow:

```
flow.tcl -design <yourdesign> -tag foo -overwrite
```

The RTL description is synthesized to a SPICE netlist containing SKY130 standard cells, the netlist is saved to `runs/foo/results/signoff/<yourdesign>.gds.spice`. The generated file can be further converted to a `.xspice` netlist using the Python script `spi2xspice.py` from `qflow`. This process inserts an interface between the analog and digital modules, they are called `toana_1v8` and `todig_1v8`. Simultaneously, the analog standard cell subcircuits are replaced with abstracted behavioral models. Unfortunately, there are no individual behavioral models for SKY130 standard cells available at the moment, instead, a generalized timing model is specified for all cells. The timings are specified in the command line options, and the values have been chosen based on a recommendation in the `open-source silicon.dev` Slack channel [22].

```
python3 spi2xspice.py \
$PDKPATH/libs.ref/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib \
-io_time=500p -time=50p -idelay=5p -odelay=50p -cload=250f \
<yourdesign>.spice <yourdesign>.xspice
```

A workaround was needed regarding the `conb`[3] cells in the XSPICE module, because the used version of `spi2xspice.py` does not insert behavioral models for standard cell `conb`, thus the simulation fails. To apply this workaround, search for the `d_genlut_sky130_fd_sc_hd__conb_1` cells in the generated `.xspice` file:

```
instance1 [] [instance1/HI net1] d_genlut_sky130_fd_sc_hd__conb_1
instance2 [] [net2 instance2/LO] d_genlut_sky130_fd_sc_hd__conb_1
```

The `conb` cells have two output ports `[HI LO]`, connected nets are tied to either constant HIGH = $V_{\mathrm{DD}}$ or LOW = $V_{\mathrm{SS}}$. In this example, `instance1` sets `net1` to LOW, `instance2` sets `net2` to HIGH. These nets must be tied to a fixed value by using the digital one and digital zero models instead, `done` and `dzero`:

```
instance1 net1 dzero
instance2 net2 done
```

Additionally, the I/O interface using the models `toana_1v8` and `todig_1v8` were not always instantiated correctly. If `ngspice` outputs the warning `singular matrix` and

[2]The conversion can also be done using the `yosys` synthesizer, but this method is not described in this work.

[3]The `conb` standard cells connect digital nets to a constant HIGH or LOW value.

unpredictable simulation waveforms occur, then it may have happened that some input and output ports did not get the correct interface. Errors have to be given special attention and must be repaired. The following snippet shows an example where the digital output `col_out_15_` has been falsely converted to a digital input:

```
AD2A20 [col_out_14_] [a_col_out_14_] toana_1v8
AA2D4 [a_col_out_15_] [col_out_15_] todig_1v8
```

The `.xspice` netlist can be included in an `xschem` schematic using the `.include` command. Instances of the digital subcircuit can be placed using a regular `xschem` symbol with matching I/O port order and the symbol property `type=primitive`. Analog instances can be placed in the same schematic for the proposed mixed-signal simulation. Note that the SPICE syntax is port order sensitive, it is important to ensure that the subcircuit and the symbol have the same port order. The finished mixed-signal netlist can be simulated with `ngspice`.

## 4.2 Digital Core Top-Level Module

The Verilog description of the digital core model `adc_core_digital.v` is shown in Listing A.6, and the structure of the digital core module can be seen in Fig. 4.7. This module combines the SAR control logic, the oversampling logic, and the matrix row and column decoders. As a result, the core module contains the digital implementation of the SAR-ADC, and it is the interface between the analog macros and the external signals.

The row and column decoders (instances `ndc` and `pdc`) implement the combinatorial logic to drive the DAC matrix input signals with thermometer-code activation using the sequential or synchronous scan mode shown in Fig. 4.4. The SAR control logic (`cnb`) implements the non-binary SAR algorithm with the chosen non-binary values, averaging filter for the 4 least significant samples, and it controls the enable signal for the self-clocked loop. Processing is done synchronously via the clock input port `clk`. The oversampling module (`osr`) calculates the oversampled 16-bit result, which is calculated asynchronously by using a register-buffered strobe signal as edge sensitive clock[4] instead of the digital clock signal. This method is expected to decrease total power consumption when a high LSB averaging mode is used. Asynchronous clock gating has also been used in the low-power SAR-ADC in [11] because the `osr` module is only active when a conversion is finished.

The core module is interfaced through the inputs `clock_dig`, `reset`, the comparator-result and the two 16-bit configuration words. The output signal `enable_loop_out` controls if the asynchronous clock generator is enabled. The `sample`, `pmatrix`, and `nmatrix` signals control both capacitive DACs. Signal `conv_finished_out` signalizes the end of a single conversion. An update of the obtained data on the 16-bit output port `result_out` is signalized with the `conv_finished_osr_out` signal.

[4]The result is updated one clock edge before the strobe signal to ensure data integrity.

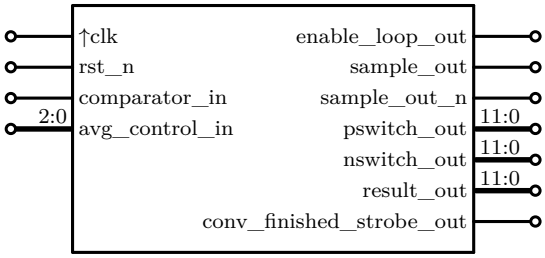



**Figure 4.1:** The block diagram of `adc_control_nonbinary.v`.

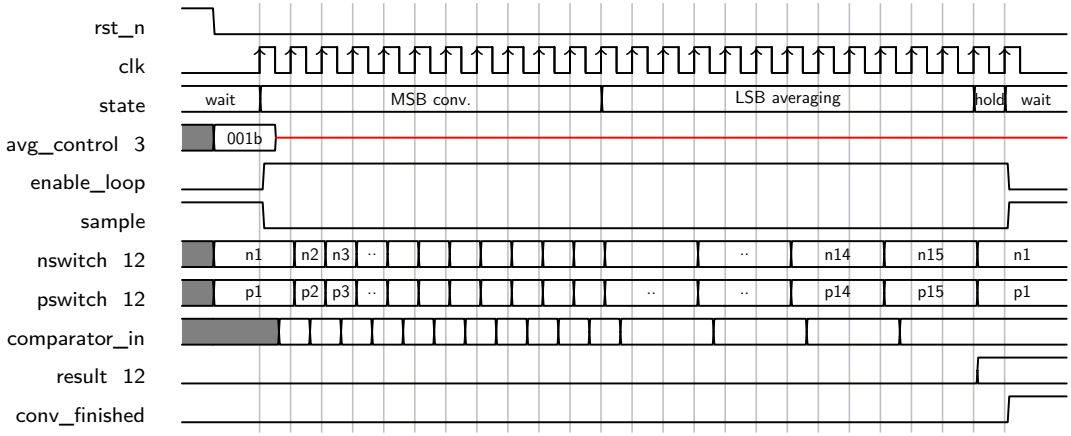



**Figure 4.2:** The timing diagram of the non-binary control and averaging module with 3 samples per averaged SAR weight.

## 4.3 Non-Binary Control and Averaging

This module implements the state machine of the SAR-ADC, calculation of the non-binary DAC value sequence, and averaging of the 4 least significant SAR weight comparison results. The structure of the non-binary SAR control and averaging module is shown in Fig. 4.1, and the HDL implementation can be seen in Listing A.7, whereas one of the possible timing diagrams is shown in Fig. 4.2.

### 4.3.1 State Machine

The state machine is implemented using a 17-bit shift register whereas each bit refers to one of the states that are described in Table 4.1. In the wait state, no digital clock is present without influence from outside, because the output signal `enable_loop_out` disables the external clock generator. At a positive clock edge, which is triggered by the top-level input signal `start_conversion`, the wait state changes to conversion mode. Simultaneously, the configuration input `avg_control_in` is stored in an internal register, whereas setting the `sample_out` and `sample_out_n` signals changes the S & H switches from sampling to hold mode. The SAR-ADC algorithm described in Sec. 1.2 is executed, whereas the calculated SAR weights are continuously written to the output ports `pswitch_out` and `nswitch_out` formatted as binary 12-bit values. Their value determines the number of activated capacitive DAC cells in the DAC matrix. State shift register bit 16 to bit 6 are referring to the most significant SAR weights where a single clock cycle is used per ADC sample. In bit 5 to bit 2, the least significant SAR weights are tested. An averaging filter is implemented for the least

**Table 4.1:** Mapping of the 17 bit in the shift register from `adc_control_nonbinary.v` to the states of the state machine.

| Bit | State | Description |
|---|---|---|
| 16–6 | MSB conversion | Process the non-binary SAR algorithm for the most significant SAR weights by evaluating the comparator value at input port `comparator_in`. |
| 5–2 | LSB averaging | Process the non-binary SAR algorithm for the least significant SAR weights by calculating the average binary value of input port `comparator_in` and evaluating the averaged result. |
| 1 | Hold result | ADC conversion is finished, this additional state ensures that the 12-bit result is valid before the next digital clock edge when the oversampling module is triggered asynchronously. |
| 0 | Wait | The module is waiting for the next positive clock edge to leave the idle state, which triggers the start of the analog-to-digital conversion. The DAC is set to sampling mode, the clock loop is blocked and the previously evaluated result is held at the output. The `conv_finished_strobe_out` port is signaling a finished ADC conversion after a conversion cycle has been completed. |

significant weights, and the number of comparator decisions to be averaged (1, 3, 7, 15, 31) can be configured by value `avg_control_in`. State bit 1 is implemented to update the 12-bit result `result_out` one clock cycle before the following strobe signal `conv_finished_strobe_out` is being set, this is done to ensure that the data on port `result_out` is valid when the `osr` module reads from it asynchronously.

### 4.3.2 Nonbinary SAR Algorithm

The algorithms for the binary and non-binary SAR weights are visualized in Fig. 1.5, whereas both algorithms start by comparing the sampled value to half of the reference voltage. This reference voltage is set by activating 1/2 of the DAC capacitors. In a binary SAR-ADC, the next comparison would be done with 1/4 or 3/4 of the reference voltage, subsequently, a SAR-ADC with $N$ bit needs exactly $N$ conversion cycles.

The proposed SAR-ADC is using SAR redundancy with non-binary weights, which introduces some error tolerance [10]. In this work, the second comparison would be done with $1/4 + 21.3\%$ or $3/4 - 21.3\%$ of the reference voltage. Both possible outcomes are calculated simultaneously, and the comparator result determines which value is multiplexed to the data register. The ratio between the SAR weights is in the range (1,2] and determines the error tolerance, i.e., a ratio of 2 relates to the binary algorithm without error tolerance. Kuttner [10] has proposed a SAR weight ratio of 1.85, and the average in Schmickl [11] was 1.83. The chosen ratio in this work was lowered to 1.65, which introduces an increased error tolerance. The value of 1.65 has

**Table 4.2:** Comparison of binary and non-binary SAR code.

| Binary | Factor | Non-binary | Factor |
|---|---|---|---|
| 2048 | — | 2048 | — |
| 1024 | — | 806 | — |
| 512 | 2 | 486 | 1,66 |
| 256 | 2 | 295 | 1,65 |
| 128 | 2 | 180 | 1,64 |
| 64 | 2 | 110 | 1,64 |
| 32 | 2 | 67 | 1,64 |
| 16 | 2 | 41 | 1,63 |
| 8 | 2 | 25 | 1,64 |
| 4 | 2 | 15 | 1,67 |
| 2 | 2 | 9 | 1,67 |
| 1 | 2 | 6 | 1,5 |
| — | — | 4 | 1,5 |
| — | — | 2 | 2 |
| — | — | 1 | 2 |
| **4095** | | **4095** | |

been chosen due to practical reasons, it was not possible to find a useful set of values for 12 bit and 3 redundant conversions. The implemented non-binary SAR code is shown in Table 4.2. The increased number of clock cycles can be compensated by raising the ADC frequency. This compensation is possible because of the decreased DAC step size in combination with the usage of a thermometer-coded DAC.

### 4.3.3 Averaging Configuration

Averaging for the comparator result at the least significant samples has been implemented to reduce the effect of comparator noise in the LSB region. The LSB region can be defined by selecting the respective state shift register bits in the `lsb_region_w` value. In this work the region of bit 2 to bit 5 has been chosen, which means that the averaging filter is applied when the least significant weights {1,2,4,6} are being evaluated:

```
wire lsb_region_w = (shift_register_r[2] | shift_register_r[3] |
↪  shift_register_r[4] | shift_register_r[5]);
```

The 1-bit comparison result `comparator_in` is summed in a 5-bit register. The number of samples to be averaged can be {1,3,7,15,31}, which is configured using the 3-bit input `avg_control_in`. The chosen numbers reduce computing complexity since only the MSB bit needs to be evaluated, e.g. 3 samples can have the 5-bit sums {0000**0**,0000**0**1,0001**0**,0001**1**}, the averaged comparator result is directly obtained by evaluating bit 1.

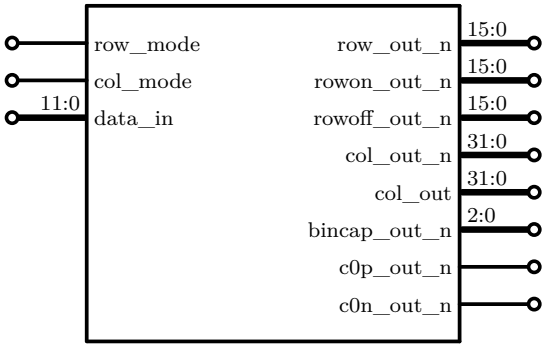



**Figure 4.3:** Block diagram of `adc_row_col_decoder.v`.

```
assign average_count_limit_w = (sampled_avg_control_r == 3'b001) ? 5'd3  :
                               (sampled_avg_control_r == 3'b010) ? 5'd7  :
                               (sampled_avg_control_r == 3'b011) ? 5'd15 :
                               (sampled_avg_control_r == 3'b100) ? 5'd31 :
                               5'd1;
```

## 4.4 Row and Column Decoder

This module is purely combinatoric and it translates a binary value at the 12-bit port `data_in` to row and column thermometer code for usage with the previously designed DAC matrix in Sec. 2.8. The symbol of this module is shown in Fig. 4.3.

A single DAC matrix includes 4096 capacitors: A single capacitor, binary capacitor cells with 1+2+4=7 capacitors, and 511 thermometer-coded capacitor cells with 8 capacitors each. The binary-coded capacitors can be directly mapped to `data_in` bit 2 to bit 0, columns are mapped from bit 7 to bit 3, and the rows from bit 11 to bit 8. The $4096^{th}$ capacitor, which can not be addressed by the data port, must be tied to a constant value.

```
port    [              data_in               ]
bit     [11][10][9][8][7][6][5][4][3][2][1][0]
mapping [    row     ][    col      ][bincap ]
```

Using a fully differential SAR ADC, both DACs must be addressed in differential directions. To define the state of the $4096^{th}$ capacitor, the constant outputs `c0p_out_n` and `c0n_out_n` have been introduced, where c0p is always on and c0n is always off. The first comparison is done with 1/2 of the reference voltage, therefore 2048 of 4096 capacitors must be active in both DAC matrices. For the n-side DAC at the comparator `inn`-port, this is done by setting 2048 to port `data_in`. For the p-side DAC on the comparator `inp` port, the differential value is obtained by a simple inversion of the n-side data. The inverted value in a 12-bit representation of 2048 is 2047 which is missing one active capacitor. An offset has been implemented using the $4096^{th}$ capacitor since 2047 plus 1 always-on capacitor also results in 2048 active capacitors.

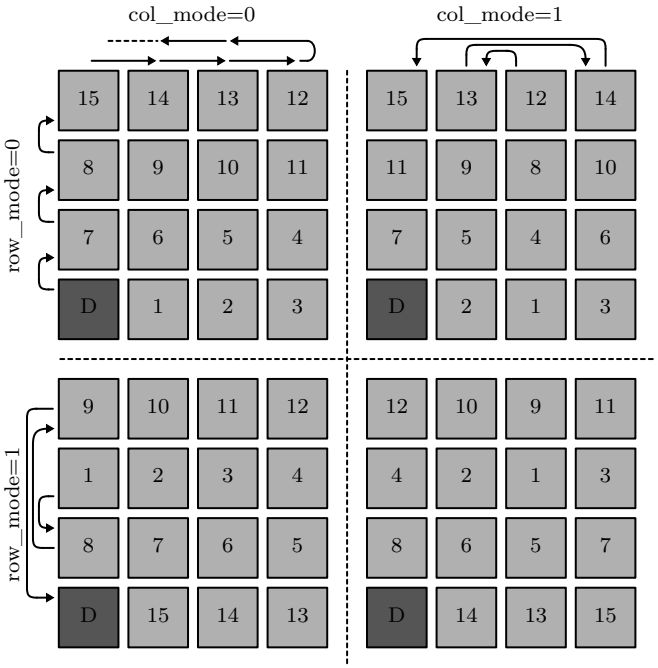



**Figure 4.4:** The implemented modes of the DAC matrix row/column decoder in dependence of the configuration bit `row_mode` and `col_mode` in a $4 \times 4$ matrix.

Alternatively, it would have been possible to add value 1 to the p-side data input, but this would have resulted in a 12-bit adder, and also the introduction of a borderline case where both DAC values can be 0 simultaneously. The final range of the n-side DAC reference voltage is therefore 0/4096 to 4095/4096, while the range of the p-side DAC is from 4096/4096 to 1/4096.

The implemented switching schemes of the row and column thermometer-code decoder are shown in Fig. 4.4. Previous work [23] has shown that a symmetrical switching mode `[6 4 2 1 3 5]` decreases the integral linearity error compared to sequential switching `[1 2 3 4 5 6]`. On the other hand, the differential error has been increased, thus the selection of the mode is a trade-off between differential and integral linearity error. As a result, both schemes have been implemented in the row and the column decoder. The configuration inputs to select the modes are `row_mode` and `col_mode` where a LOW=0 V activates the sequential mode and HIGH=1.8 V the symmetrical mode. The implementation has been done by using a shared shift register for both modes to keep hardware complexity low. As a result of using the shared shift register, the different modes can be implemented mostly by using multiplexers between the shift register and the DAC interface. Therefore, the implementation complexity is reasonably low.

## 4.5 Oversampling Module

The oversampling module asynchronously reads the data at the input port, and a boxcar filter calculates the oversampled result by adding up a configured number of samples. The block diagram of the module can be seen in Fig. 4.5.

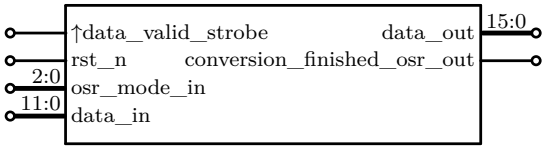



**Figure 4.5:** Block diagram of `adc_osr.v`.

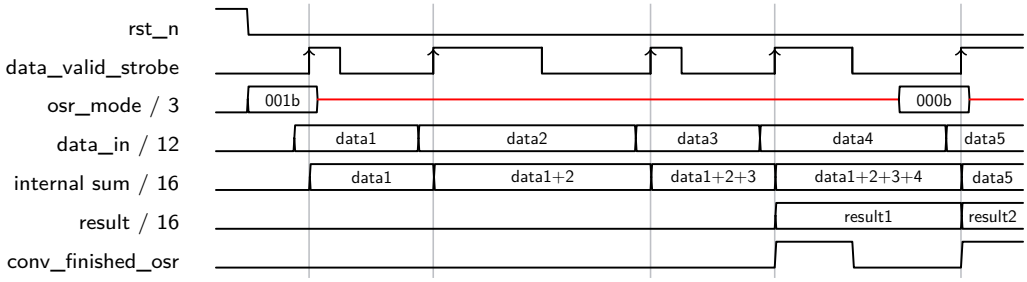



**Figure 4.6:** The timing diagram of the oversampling module showing a conversion using 4 samples per result, which is then followed by a conversion using 1 sample per result.

The result is asynchronously read without using the same clock signal as the core module, therefore it is important to make sure that the data input is valid when the signal `data_valid_strobe` is set. Additionally, the signal `data_valid_strobe` needs to be sourced directly from a flip-flop to avoid glitches due to the usage of a combinatoric signal. If everything is set up correctly, then the `osr` configuration `osr_mode_in` is read and stored at the first sample, and the configured number of samples is read on port `data_in`. Signal `conversion_finished_osr_out` signalizes a finished conversion cycle and `data_out` is updated. Note that this signal is asynchronously linked to `conversion_finished_out`.

Oversampling is done due to the nature of the quantization noise. The effective number of bits (ENOB) in an ideal ADC can be increased by improving the signal-to-quantization noise ratio. As a rule of thumb, the ENOB for a sinusoidal signal is increased by 1 bit (6.02 dB/bit) if the number of samples is multiplied by $N = 4$ ($10 \log N$). Input signal `osr_mode_in` is used to set the number of samples for oversampling, up to 256 samples (+4 bit ENOB) have been implemented.

```
//   mode 001 = 4 samples   +1 Bit resolution
//   mode 010 = 16 samples  +2 Bit resolution
//   mode 011 = 64 samples  +3 Bit resolution
//   mode 100 = 256 samples +4 Bit resolution

// Amount of oversampling samples (+N Bit SNR per 4^N samples)
wire [8:0] osr_count_limit_w = (osr_mode_r == 3'b001) ? 9'd4   :
                               (osr_mode_r == 3'b010) ? 9'd16  :
                               (osr_mode_r == 3'b011) ? 9'd64  :
                               (osr_mode_r == 3'b100) ? 9'd256 :
                               9'd1;
```

The result, which is written to output port `data_out`, is aligned to the MSB. For example, a single 12-bit sample is shifted left in the result to `{sum[11:0],4'b0}`. If

oversampling is used, then the increased resolution is reflected in the 4 bit LSB. Using 4 samples, the sum length is increased by 2 bit and the effective resolution by 1 bit. The oversampled value is then mapped to `data_out` by `{sum[13:1],3'b0}`, where discarding bit 0 equals a division by 2. This scheme continues with `{sum[15:2],2'b0}` for 16 samples, `{sum[17:3],1'b0}` for 64 samples, and `{sum[19:4]}` for 256 samples. The full-scale range (FSR) of the ADC is mapped from the result vector `16'h0000` to the maximum value of `16'hFFF0`.

```
assign next_output_w = bypass_oversampling  ? {next_result_w[11:0],4'b0} :
                       ~is_last_sample      ? output_r                   :
                       (osr_mode_r==3'b001) ? {next_result_w[13:1],3'b0} :
                       (osr_mode_r==3'b010) ? {next_result_w[15:2],2'b0} :
                       (osr_mode_r==3'b011) ? {next_result_w[17:3],1'b0} :
                       (osr_mode_r==3'b100) ? {next_result_w[19:4]}      :
                       16'bX;
```

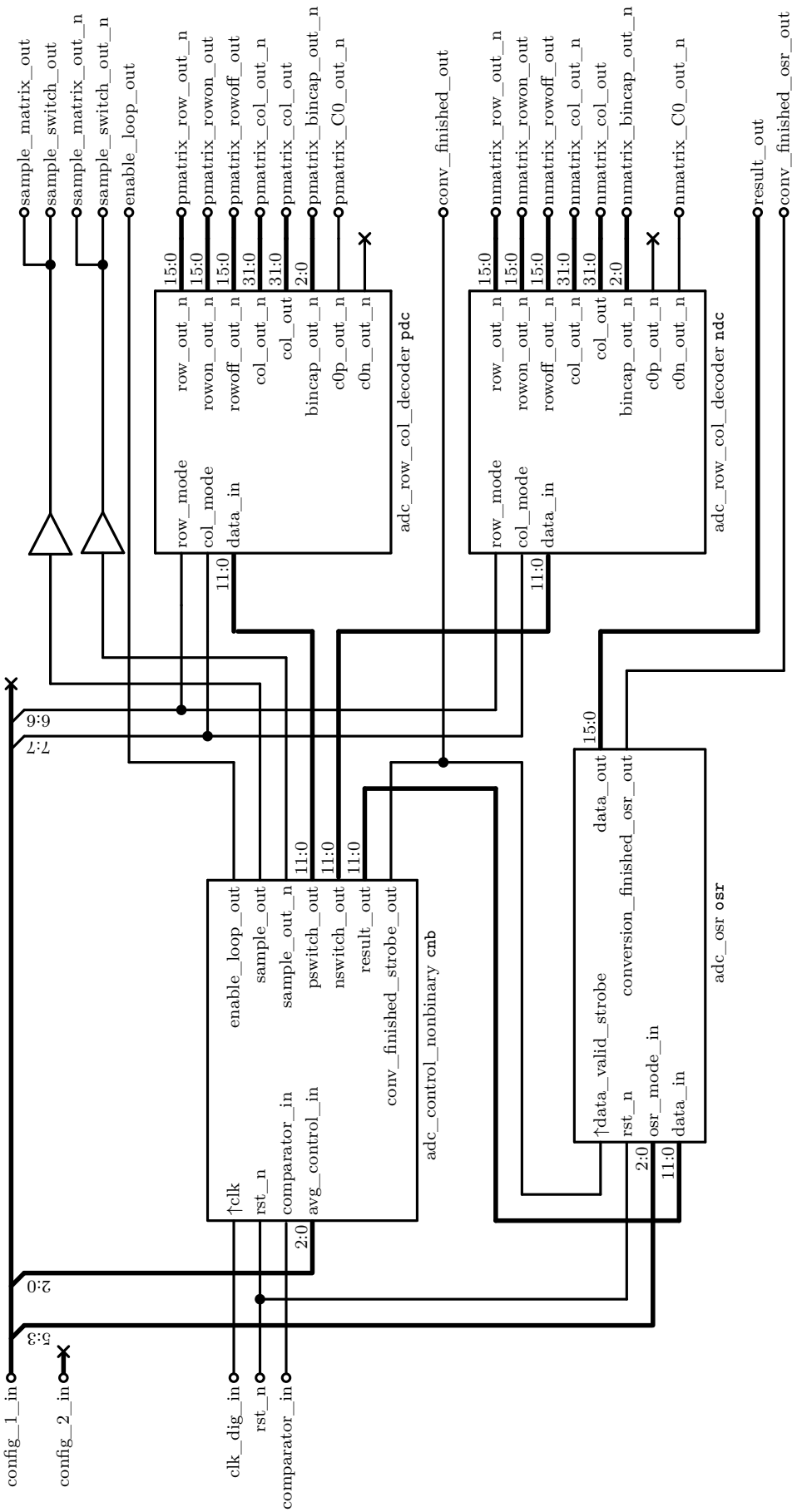



**Figure 4.7:** Block diagram of `adc_core_digital.v`.

# Chapter 5

# Top-Level

The mixed-signal top-level design combines analog macros with digital logic. First, the interface between the digital logic and the analog macros is defined using Verilog HDL, and also the I/O interface is specified in a Verilog top-level module. At this stage, a mixed-signal simulation can be done before the layout is generated. When the RTL2GDSII flow `OpenLane` is configured, then the design can be hardened. The result is the `.gds` and `.lef` file of the top-level SAR-ADC, which can be integrated into a chip. Finally, a post-layout parasitic extraction can be performed using `magic`, which is followed by a post-layout simulation.

The following list summarizes the current (03/2023) sources used for the top-level design toolset in this work:

- SKY130: Open-PDKs installer (`github.com/RTimothyEdwards/open_pdks`).
- magic VLSI layout tool: Analog layouts with pcell generation, DRC check, parasitic extraction and post-layout SPICE-file generation (`github.com/RTimothyEdwards/magic`).
- xschem: Electrical circuit simulator with internal waveform viewer and mixed-signal support (`github.com/StefanSchippers/xschem`).
- ngspice: Mixed-signal simulator (`github.com/ngspice/ngspice`).
- gaw3: Waveform viewer (`github.com/StefanSchippers/xschem-gaw`).
- OpenLane: Automated RTL2GDSII flow (`github.com/The-OpenROAD-Project/OpenLane`).
- Xyce:High-performance parallel simulator (`xyce.sandia.gov`).

## 5.1 Top-Level Overview

The top-level SAR-ADC symbol with its I/O interface is shown in Fig. 5.2. The module is interfaced by the input ports of the $V_{\mathrm{CM}}$ generator clock (32.768 kHz), the conversion start signal, both analog differential inputs, and 16-bit configuration ports.

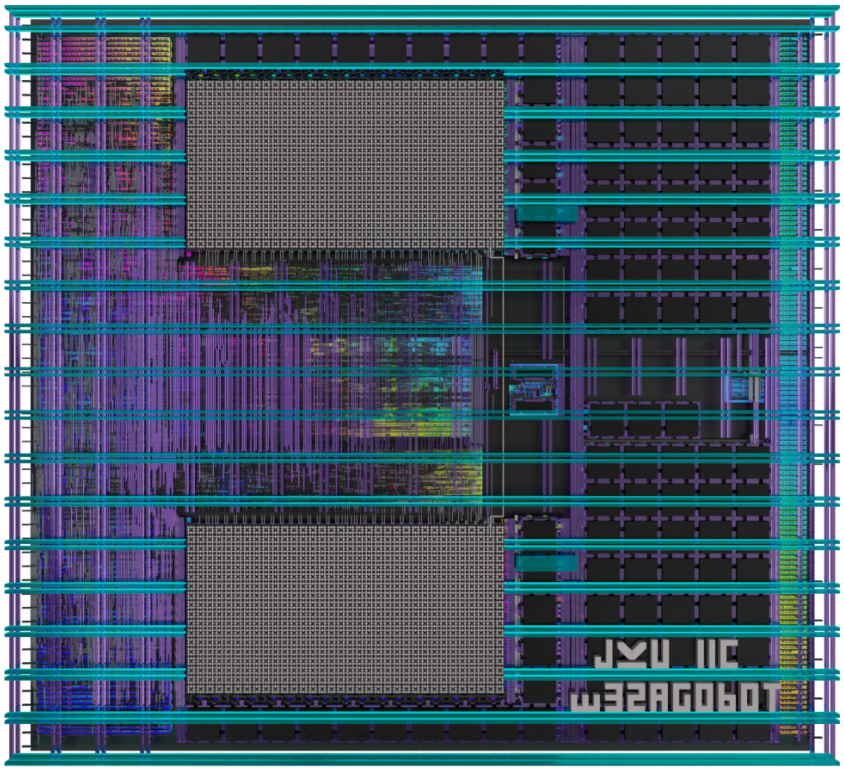


**Figure 5.1:** 3D render of the ADC top-level layout.

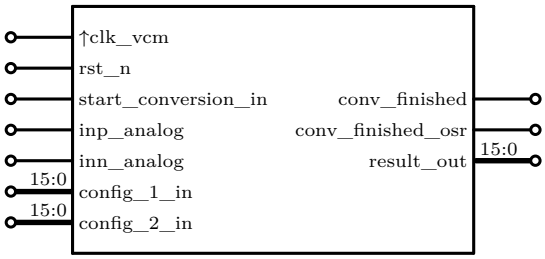



**Figure 5.2:** Symbol of the SAR-ADC top-level module `adc_top`.

The outputs include the 16-bit result port and the conversion-finished strobes. A detailed description of the configuration ports can be seen in Table 5.1 and Table 5.2.

The top-level schematic is shown in Fig. 5.13. The `adc_top` module combines the digital RTL logic `adc_core_digital.v` with the analog macrocells. In the schematic, the digital core is visualized with the color blue, and the analog macros are shown in green. Analog wires need to be treated differently than digital signals, they are marked as red and black dotted lines. Note that the sample matrix signal is not directly routed to the DAC matrices, instead, it is routed through the clock generator where a delay chain is integrated. This measure ensures that the sample switch signal is turned off before the sample matrix signal becomes active. The layout is intended to be hardened using the automated RTL2GDS flow `OpenLane`, which is based on a

**Table 5.1:** Pin description of the 16-bit configuration port `config_1_in`.

| Name | Bit | Value | Modifier |
|---|---|---|---|
| Averaging filter | 2..0 | `000` | no averaging (default) |
| | | `001` | 3 samples |
| | | `010` | 7 samples |
| | | `011` | 15 samples |
| | | `100` | 31 samples |
| Oversampling factor | 5..3 | `000` | no oversampling (default) |
| | | `001` | 4 samples |
| | | `010` | 16 samples |
| | | `011` | 64 samples |
| | | `100` | 256 samples |
| Row mode | 6 | `0` | sequential |
| | | `1` | symmetric |
| Column mode | 7 | `0` | sequential |
| | | `1` | symmetric |
| Unused | 9..8 | xx | |
| Edge detect delay | 15..10 | `xxxxx1` | +5 ns delay |
| | | `xxxx1x` | +10 ns delay |
| | | `xxx1xx` | +20 ns delay |
| | | `xx1xxx` | +40 ns delay |
| | | `x1xxxx` | +80 ns delay |
| | | `1xxxxx` | +160 ns delay |

Verilog HDL description of the circuit. Therefore, the schematic has been described with Verilog HDL as seen in Listing A.10.

## 5.2 Mixed-Signal Simulation

The functional integrity of the top-level design has been verified in a mixed signal simulation before the top-level layout is generated. The simulation of analog circuits is compute-intensive, and to ease the requirements the digital RTL logic `adc_core_digital.v` has been converted to a behavioral model in XSPICE format `adc_core_digital.v.xspice`. This behavioral model was combined with the analog macros in the testbench `adc_top_tb.sch`, whereas the mixed-signal circuit can be simulated using `ngspice`. The process to convert the RTL description to the behavioral netlist with analog interfaces has been described in Sec. 4.1.2. The `xschem` schematic of the testbench can be seen in Fig. A.1 and Fig. A.2.

The simulation results at a differential input voltage of $V_{\text{diff}} = V_{\text{inp}} - V_{\text{inn}} = 200\,\text{mV}$ have been plotted in Fig. 5.3. `ctopp` and `ctopn` are the voltages on the DAC capacitor top plates, and the comparator evaluates which of those is at a higher potential. `clk_dig` is the digital clock signal from the clock generator macro, and `clk_comp` is the comparator clock. The conversion is started with a positive edge at signal `start_conv`, and the clock generator is then held active for 17 clock cycles[1] until one conversion is done. The evaluated result of this run, shown in Listing 5.1, was

[1]The number of clock cycles can change with different averaging and oversampling configurations.

**Table 5.2:** Pin description of the 16-bit configuration port `config_2_in`.

| Name | Bit | Value | Modifier |
|---|---|---|---|
| Delay 1 | 4..0 | `xxxx1` | +5 ns delay |
| | | `xxx1x` | +10 ns delay |
| | | `xx1xx` | +20 ns delay |
| | | `x1xxx` | +40 ns delay |
| | | `1xxxx` | +80 ns delay |
| Delay 2 | 9..5 | `xxxx1` | +5 ns delay |
| | | `xxx1x` | +10 ns delay |
| | | `xx1xx` | +20 ns delay |
| | | `x1xxx` | +40 ns delay |
| | | `1xxxx` | +80 ns delay |
| Delay 3 | 14..10 | `xxxx1` | +5 ns delay |
| | | `xxx1x` | +10 ns delay |
| | | `xx1xx` | +20 ns delay |
| | | `x1xxx` | +40 ns delay |
| | | `1xxxx` | +80 ns delay |
| Delay control | 15 | `0` | Configuration disabled, all delays set to the maximum value |
| | | `1` | Configuration enabled, delays use the configured values |

**Listing 5.1:** The terminal output of the result calculation at the end of the transient simulation.

```
let k = length(time) - 1
(((result15[k]*2048 + result14[k]*1024 + result13[k]*512 +
↪  result12[k]*256 + result11[k]*128 + result10[k]*64 + result9[k]*32 +
↪  result8[k]*16 + result7[k]*8 + result6[k]*4 + result5[k]*2 +
↪  result4[k]*1 + result3[k]*0.5 + result2[k]*0.25 + result1[k]*0.125 +
↪  result0[k]*0.0625)-2048*1.8)/2048) = 2.021484e-01
```

$V_{\text{result}} = 202.15\,\text{mV}$. The difference $V_{\text{result}} - V_{\text{diff}} = 2.15\,\text{mV}$ can be justified by the matching mismatch in the binary DAC capacitor cells and a possible offset error. Delay settings have been `00010` (10 ns) for `delay1`, `delay2`, and `delay3`. The edge detection delay has been set to `001100` (60 ns). These settings resulted in a digital clock frequency of $f_{\text{clk\_dig}} = 18.10\,\text{MHz}$. With averaging and oversampling disabled, the total time from the start signal to the result output was $T_{\text{conv}} = 912.23\,\text{ns}$. Assuming a settling time of 20 ns, this results in a sampling rate of $(T_{\text{conv}} + 20\,\text{ns})^{-1} = 1.072\,\text{MS/s}$.

## 5.3 Macro Hardening

The SAR-ADC macro is hardened using the `OpenLane` flow. The configurations set in the `OpenLane` configuration file `config.json` are shown in Listing A.11. First, the floor plan is defined including macro placement, pin placement, and the power and standard cell grid generation. A synthesis strategy has been chosen, which is followed by executing the `OpenLane` hardening process. The flow outputs a `.gds` and `.lef` file of the SAR-ADC macro which can be integrated into another design.

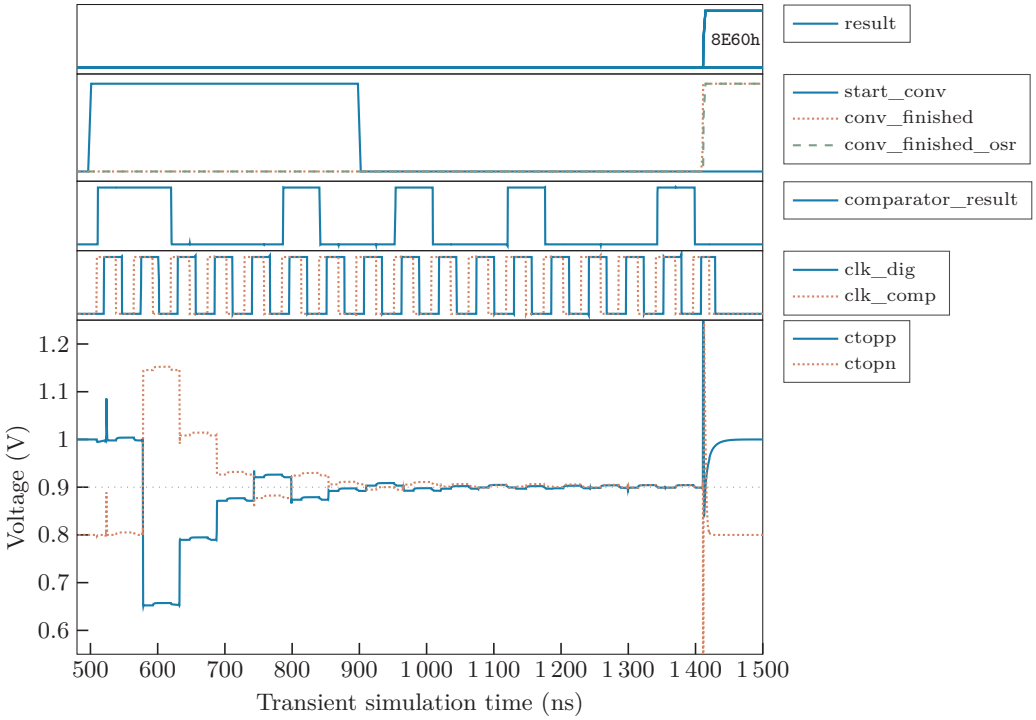



**Figure 5.3:** Pre-layout mixed-signal simulation of the top-level. The differential voltage $V_{\text{inp}} - V_{\text{inn}}$ has been set to 200 mV.

### 5.3.1 Manual Macro Placement

The floorplan for this design has been inspired by the layout seen in Fig. 1.6 [9]. The RTL2GDSII flow `OpenLane` allows to either let the flow place the macros automatically or manually by specifying the macro coordinates. In this design, the manual method has been chosen. Setting the configuration `FP_SIZING` to absolute, in addition to the fixed chip area `DIE_AREA`, defines the outlines of the die. The configuration file at path `MACRO_PLACEMENT_CFG` is used to specify where the macros should be placed on the die. The `.lef` files, `.v` black boxes, and `.gds` layouts of the macros need to be specified in the configuration variables `VERILOG_FILES_BLACKBOX`, `EXTRA_LEFS` and `EXTRA_GDS_FILES`. The paths to the top-level and the RTL Verilog files are configured with `VERILOG_FILES`. The result after configuration of the manual macro placement in `OpenRoad` is shown in Fig. 5.4.

### 5.3.2 Power Grid Generation

The free area, which is not occupied by the macros, is filled with the digital standard cell grid. This area is used for the placement of the standard cells from the synthesized digital logic. In this design, the high-density standard cell library `sky130_fd_sc_hd` has been chosen. This library is specified with configuration `STD_CELL_LIBRARY`. Furthermore, the pin positions are placed in the design according to the configured file in `FP_PIN_ORDER_CFG`.

The current `OpenLane` methodology [21] differentiates between the core, macro, and pad frame hardening. First, the macros are hardened using up to `met4`. Macros inside


**Figure 5.4:** `OpenRoad` manual macro placement stage of `adc_top`.

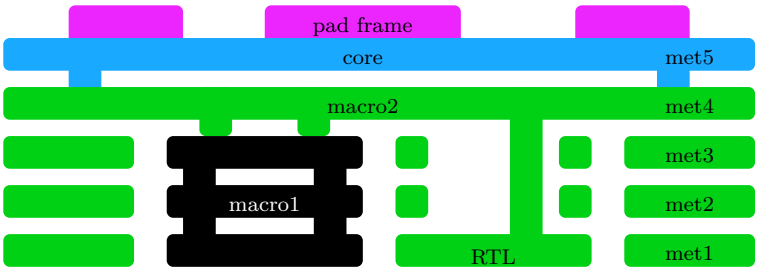



**Figure 5.5:** Hardening hierarchy used in `OpenLane`. The different hierarchical levels are color coded. Black is a hardened macro nested inside of the green macro. The green macro is integrated into the blue core hierarchy, and purple represents the pad frame level.

of a macro are powered over the vertical `met4` power distribution network (PDN). As a result, macros inside the macro should contain horizontal power rails on `met3`, macros inside this macro vertical `met2` rails, and so on. The topmost macros are intended to be integrated into the core, where macros are hooked up to the horizontal PDN on `met5`. The core is then integrated into the pad frame using one further metal layer above `met5`. The hardening hierarchy is visualized in Fig. 5.5.

The `OpenLane` documentation specifies:

> "..in the skywater pdk the metal stack has 5 layers, thus for the core level you can use all layers up to met5; however, if you have another macro inside your core, that macro can only use up to met4, and so forth." - `github.com/The-OpenROAD-Project/OpenLane/blob/master/docs/source/usage/advanced_power_grid_control.md`

The design in this work is bending this rule a bit, the used hierarchy can be seen in Fig. 5.6. A different workflow has been chosen because the "correct" workflow restricts the SAR-ADC to use metal layers up to maximum `met4`, and macros inside the macro

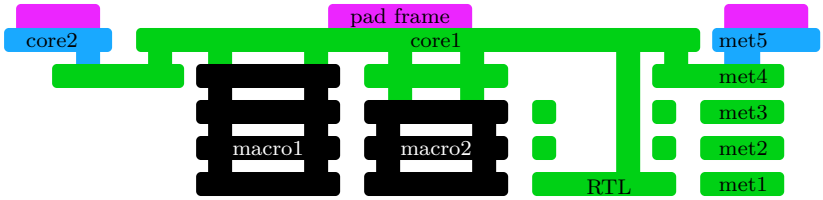



**Figure 5.6:** Hardening hierarchy used in this work. The hierarchical hardening levels are color coded. Hardened macros are black, they are integrated into the green area which is hardened as a core. The hardened core `core1` is then added to the blue-colored $2^{nd}$ core level `core2`. Purple represents the pad frame level.

would be restricted to maximum `met3`. This makes the use of MIM capacitors inside nested macros impossible. To overcome this limitation, a method has been found to integrate macros on the same hierarchical level. As a result, the SAR-ADC macro is hardened as a "core" which is intended to be integrated into another core.

Evaluation of the PDN generation behavior has revealed that it is possible to integrate a macro into a macro at the same hierarchical level. As a result, a "core" using `met5` can be integrated into another core using `met5`, unfortunately, with the following limitations: First of all, the PDN generator reads the data in the `.lef` files looking for pin and obstruction layers. Obstruction layers will always be avoided by the PDN generator. Power pins on the same layer as the generated PDN must have the same power domain. The respective power nets must be specified in configuration `FP_PDN_MACRO_HOOKS` to be recognized and properly connected. The generated macro must contain power rails one hierarchical level below the core PDN, otherwise, the PDN can not connect to the specified pins in `FP_PDN_MACRO_HOOKS`. Any other net layer in the `.lef` file will be ignored by the PDN generator, this causes shorts and DRC/LVS errors.

When the SAR-ADC macro is hardened, the routing on layer `met5` must be avoided. The routing layer of the SAR-ADC has been limited to the maximum layer `met4` by setting `RT_MAX_LAYER`. This ensures only the power distribution network is generated on layer `met5` containing exclusively `VDD` and `VSS`. Net `vcm` has been manually routed over `met5`, and obstruction layers have been added manually in the `.lef` file to prevent DRC/LVS errors. These settings allow integration of the SAR-ADC macro into another core using `OpenLane`, even if the macro contains structures on `met5`.

Furthermore, a custom PDN generation script for `OpenRoad` was necessary. The standard workflow expects macros to be connected to the PDN with integrated `met4` power rails. The clock generator macro `cgen` includes horizontal power rails on layer `met3` while `met4` is unused. Therefore, the rule shown in Listing 5.2 for powering the instance `cgen` has been added. The path to the modified script has been specified in the configuration variable `PDN_CFG`.

The `OpenRoad` power grid generation is limited to rectangular designs [24]. The PDN is generated using the area of the standard cell grid, therefore it is expected that this grid is reaching the chip edges to span an area filling the whole chip. As seen in Fig.5.4, the standard cell grid was originally planned to start at the left border and end at the

**Listing 5.2:** Custom PDN script for connection of macros to power rails on layer `met3`

```
# Custom Macro PDN-Grid
# PDN Grid exclusive setting for instance "cgen" which is powered from
↪  metal3 to metal4
define_pdn_grid -macro -name macro_cgen -instances "cgen" -starts_with
↪  POWER -halo "$::env(FP_PDN_HORIZONTAL_HALO)
↪  $::env(FP_PDN_VERTICAL_HALO)"

add_pdn_connect -grid macro_cgen -layers "met3 met4"
```

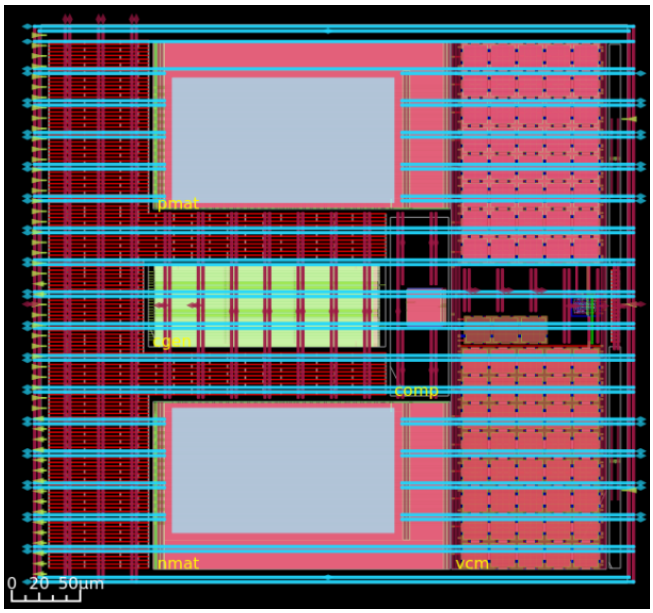



**Figure 5.7:** `OpenRoad` PDN and standard cell power grid after power distribution network generation on design `adc_top`.

comparator. As a result, the PDN was not generated over the whole chip, leaving some macros unpowered. The chip dimensions have been extended to the right, which allows the generation of a narrow standard cell grid at the right border. This ensures the generation of the PDN with a core ring covering the whole chip, and all macros can be connected to the PDN. Furthermore, obstruction macros have been placed to limit the size of this additional standard cell grid on the top and bottom, this prevents standard cell placement in the obstructed areas. Additionally, the endcap cell in configuration `FP_ENDCAP_CELL` has been changed to a tap cell `sky130_fd_sc_hd__tapvpwrvgnd_1`. This change allows the additional standard cell grid to become narrower because tap cell placement is ensured. If no tap cells would be placed, then LVS errors can occur. The resulting power distribution network with the generated standard cell grid can be seen in Fig. 5.7.

### 5.3.3 Synthesis Exploration

Fast circuits tend to be large, while slower circuits usually save area. By invoking the command `flow.tcl -design adc_top -synth_explore` the toolchain starts to evaluate different synthesis strategies with a focus on area or speed. The report shown

**Table 5.3:** Synthesis exploration results from `OpenLane`. The best results have been marked with bold text.

| Strategy | Gate Count | Area ($\mu m^2$) | Delay (ps) | Gates Ratio | Area Ratio | Delay Ratio |
|---|---|---|---|---|---|---|
| DELAY 0 | 1443 | 11489.77 | 3251.73 | 1.252 | 1.328 | 1.352 |
| DELAY 1 | 1440 | 11358.39 | 3119.24 | 1.250 | 1.312 | 1.297 |
| DELAY 2 | 1388 | 10976.78 | 3549.28 | 1.204 | 1.268 | 1.476 |
| DELAY 3 | 1439 | 11597.37 | 3646.93 | 1.249 | 1.340 | 1.517 |
| DELAY 4 | 1535 | 11835.10 | 4103.75 | 1.332 | 1.368 | 1.707 |
| AREA 0 | **1152** | **8650.80** | 5091.20 | 1.000 | 1.000 | 2.118 |
| AREA 1 | 1214 | 9072.45 | 5843.12 | 1.053 | 1.048 | 2.431 |
| AREA 2 | 1225 | 9278.90 | 5009.51 | 1.063 | 1.072 | 2.084 |
| AREA 3 | 1685 | 11320.86 | **2403.44** | 1.462 | 1.308 | 1.000 |

in Fig. 5.3 helps to decide on which synthesis strategy to choose. In this work, the best area would be achieved with strategy `AREA 0`, and the best speed with `AREA 3` while the used area is still average. As a result, the configuration `SYNTH_STRATEGY` in `config.json` has been set to `AREA 3`. The gate count is 1685 with an area of 11 320.86 $\mu m^2$. The maximum delay is 2.40 ns at the selected `CLOCK_PERIOD` of 20 ns.

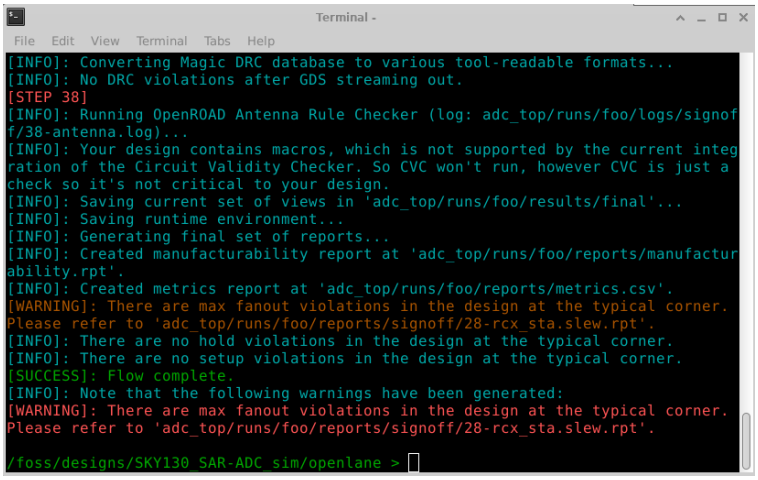
## 5.4 Clock Tree Synthesis

The RTL logic is sourced with a clock signal from a clock generator macro. Theoretically, it is best to pre-harden the RTL in a separate macro, to ensure that clock tree synthesis (CTS) is performed in the digital logic. Afterwards, only an interconnect wire between both macros is needed, the macro clock input is tied to the signal output of the clock generator.

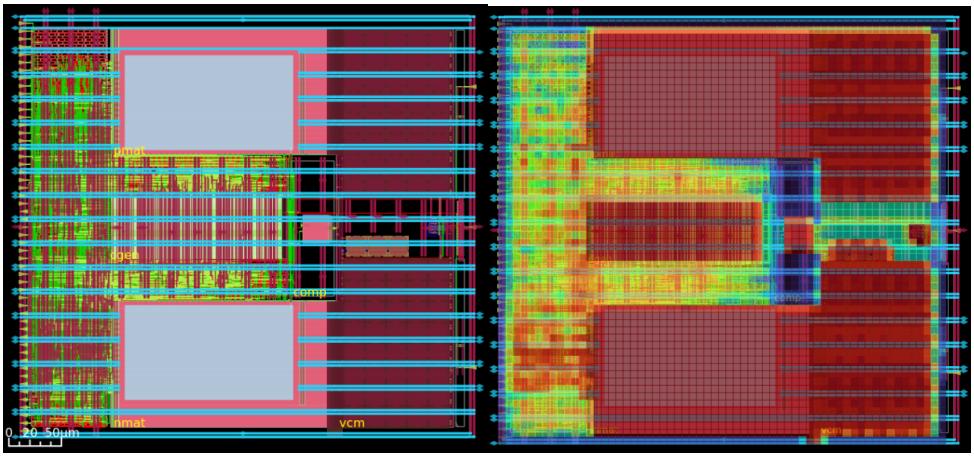
In this work, the RTL and the clock generator were hardened on the same hierarchical level to get an area-efficient design with a non-rectangular standard cell grid. No solution has been found to automatically connect both nets while clock tree synthesis (CTS) is executed on the clock net. As a workaround, a dummy clock input pin has been defined on the top-level of the SAR-ADC `clk_dig_dummy`. This pin has been specified as digital clock input in configuration `CLOCK_PORT` where clock tree synthesis and static timing analysis are performed. To finalize the SAR ADC, post-processing is needed to remove the dummy clock pin and to connect the clock generator to the clock net. This has been done using `klayout`.

## 5.5 Top-Level hardening

Hardening of the SAR-ADC macro in `OpenLane` has been started by executing the command line `flow.tcl -design adc_top -tag foo -overwrite` in the terminal. The flow is then hardening the SAR-ADC macro using the configurations from Listing A.11. The layout can be explored using the `OpenRoad` GUI by executing `flow.tcl -design adc_top -tag foo -gui`. The different views of the finished design are shown in Fig. 5.8. The resulting layout is dumped in the folder




**(a)**




**(b)** **(c)**

**Figure 5.8:** (a) `OpenLane` terminal when the flow is completed. (b) The layout view in `OpenRoad` after the routing of the design. (c) Routing heat map showing congested areas.

`adc_top/results/final/gds/adc_top.gds`. After manual post-processing of the `.gds` and `.lef` files using `klayout` (connection of $V_{\mathrm{CM}}$ from the generator to the DAC matrices, re-wiring of `clk_dig` from the clock generator, cleaner and thicker routing of the analog differential wires) the SAR-ADC macro is completed. At this point, post-layout simulations can be started to validate and, if needed, to further optimize the design.

## 5.6 Post-Layout Simulation

The layout has been verified in a post-layout simulation in a typical setting. The capacitive parasitics of the SAR-ADC GDSII layout were extracted using the extraction functionality of `magic`. With the settings shown in Listing 5.3, the result is a SPICE netlist `adc_top.gds.C.noD.merge.postlayout.spice` with 103 k devices including 76 k capacitors. Due to the high number of devices and computing effort, the parallel simulator `Xyce` was chosen for simulation. Two different test benches have been set

**Listing 5.3:** Parasitic C extraction in `magic` with limitation to capacitors $\geq 0.1\,\mathrm{fF}$.

```
gds read adc_top
load adc_top
snap internal
select top cell
flatten -dotoplabels adc_top_flat
load adc_top_flat
cellname delete adc_top
cellname rename adc_top_flat adc_top
select top cell
extract do local
extract unique
extract warn no fets
extract all
ext2spice short resistor
ext2spice lvs
ext2spice cthresh 0.1
ext2spice merge conservative
ext2spice -F
```

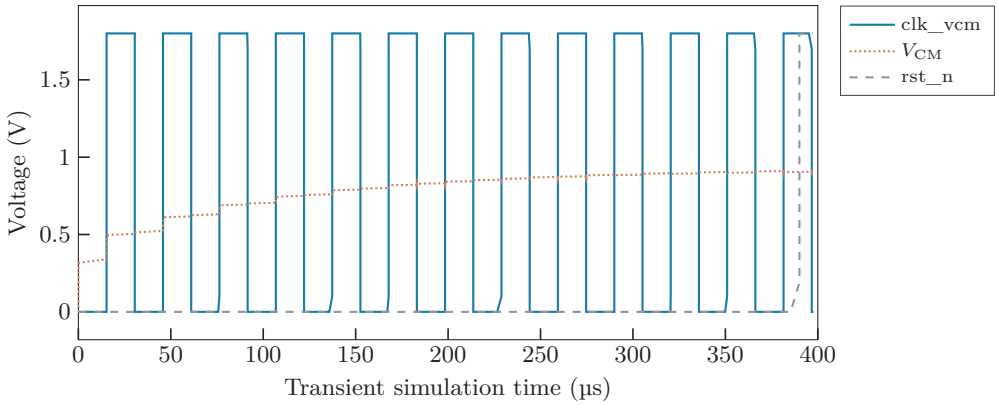



**Figure 5.9:** $V_{CM}$ generation at system boot where all voltage initial conditions have been set to 0 V at the start of the simulation.

up: `adc_top_postlayout_tb.sch` with $V_{CM}$ sourced from the generator macro, and test bench `adc_top_postlayout_tb_conv.sch` where $V_{CM} = 0.9\,\mathrm{V}$ is fixed.

The following conversion, with 4 samples oversampling and 3 samples LSB averaging, took over 25 hours of CPU time on a 16 core/32 thread Ryzen 3950x to complete. In this simulation, `ABSTOL` was set to $10^{-8}$ and `RELTOL` to $10^{-4}$. With eased tolerance settings, the comparator tends to randomly become stuck, which is not expected to happen in reality because of the naturally occurring noise.

The $V_{CM}$ reference voltage generation in the post-layout simulation is shown in Fig. 5.9. At a $V_{CM}$ generator clock of 32 768 Hz, the SAR-ADC needs 400 µs startup time until the reset signal can be released and the first conversion can be triggered.

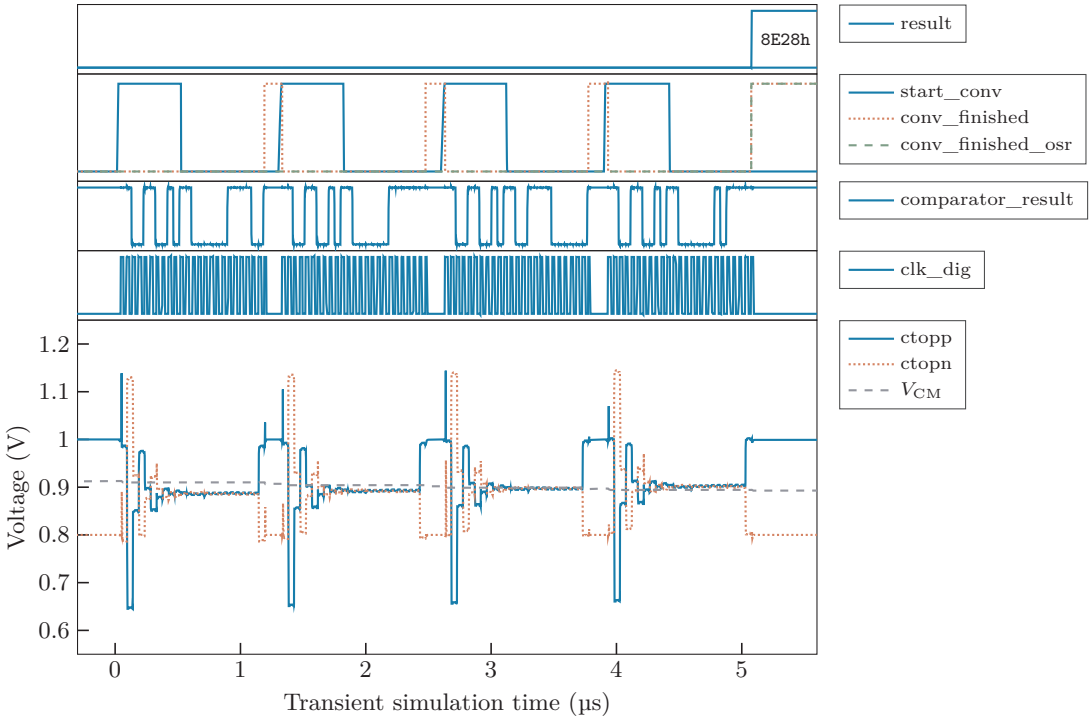



**Figure 5.10:** Post-layout simulation of a single conversion with parasitic capacitors, averaging using 3 samples, and oversampling of 4 conversions. The differential voltage $V_{\text{inp}} - V_{\text{inn}}$ has been set to 200 mV.

Fig. 5.10 shows the simulated conversion result at an input differential voltage of $V_{\text{inp}} - V_{\text{inn}} = 200\,\text{mV}$. Without offset and gain error compensation, the evaluated result `8E28h` shown in Fig. 5.10 translates to 199 mV using the formula in Listing 5.1.

The delay settings have been set to the fastest possible configuration, which is `00001` (5 ns) for `delay1`, `delay2` and `delay3`. The edge detection delay has been set to `000110` (30 ns). The evaluated digital clock period was $T_{\text{clk\_dig}} = 47.77\,\text{ns}$, whereas the frequency $f_{\text{clk\_dig}}$ is $1/T_{\text{clk\_dig}} = 20.96\,\text{MHz}$. The total number of digital clock cycles $N_{\text{clk\_dig}}$ for one conversion is calculated by $N_{\text{clk\_dig}} = 13 + 4 \cdot N_{\text{avg}}$. With $N_{\text{avg}} = 3$ samples per average (avg setting `001`) and $N_{\text{osr}} = 4$ samples oversampling (OSR setting `001`), one complete conversion needs 4 conversions with $N_{clk} = 25$ digital clock cycles per conversion. $T_{\text{settling}} = 20\,\text{ns}$ has been added to the equation to reflect the signal settling time between analog-to-digital conversions. The calculated sample rate for this mode of operation is therefore:

$$f_s = \frac{1}{N_{\text{clk\_dig}} \cdot T_{\text{clk\_dig}} + T_{\text{settling}}} = \frac{1}{25 \cdot 47.77\,\text{ns} + 20\,\text{ns}} = 823.55\,\text{kS/s} \tag{5.1}$$

The Nyquist bandwidth is determined by halving the sample rate due to the Nyquist theorem, and a division by the oversampling factor:

$$BW = \frac{1}{2 \cdot N_{\text{osr}}} \cdot f_s = \frac{1}{2 \cdot 4} \cdot \frac{1}{25 \cdot 47.77\,\text{ns} + 20\,\text{ns}} = 102.94\,\text{kHz} \tag{5.2}$$

At the fastest mode of operation, where averaging and oversampling are both disabled ($N_{\text{avg}} = 1$, $N_{\text{osr}} = 1$), the number of clock cycles per conversion is reduced to $N_{\text{clk\_dig}} = 17$ clock cycles. The fastest possible sample rate is:

$$f_s = \frac{1}{17 \cdot 47.77\,\text{ns} + 20\,\text{ns}} = 1.203\,\text{MS/s} \tag{5.3}$$

If no oversampling is done, the Nyquist bandwidth is simply half of the sample rate:

$$\text{BW} = \frac{f_s}{2} = \frac{1.203\,\text{MS/s}}{2} = 601.5\,\text{kHz} \tag{5.4}$$

The power consumption of the SAR-ADC has been evaluated for the typical setting with 3 samples averaging and 4 conversions for oversampling. The simulation data of the power consumption is shown in Fig. 5.12 with maximum, average, and minimum values calculated for 20 transient simulation results each (no uniform timestep). In the timespan of $t_{\text{conv}} = 5.05\,\mu\text{s}$ a total energy of $E_{\text{conv}} = 1.69\,\text{nJ}$ has been used, whereas the average power dissipation was $P_{\text{avg}} = 334.97\,\mu\text{W}$.

Furthermore, the power dissipation has been evaluated for the slowest conversion mode, which resembles the low-power biosignal sensor application. The slowest digital clock rate (delay setting `11111`) has been $f_{\text{clk\_dig}} = 1.016\,\text{MHz}$. The averaging mode has been set to $N_{avg} = 31$ samples per least significant weight. As a result, the number of clock cycles per conversion is $N_{clk\_dig} = 137$. With one clock cycle settling time, the sampling rate is $f_s = f_{\text{clk\_dig}}/(137 + 1) = 7.36\,\text{kS/s}$ and the Nyquist bandwidth is $BW = 7.36\,\text{kS/s}/(2 \cdot 256) = 0.014\,\text{kHz}$. Because of the high simulation complexity at higher `ABSTOL` and `RELTOL` precision, only 54 clock cycles have been simulated to save time, and the LSB power dissipation has been extrapolated. The resulting power dissipation for one conversion was $P_{\text{avg}} = 68.16\,\mu\text{W}$.

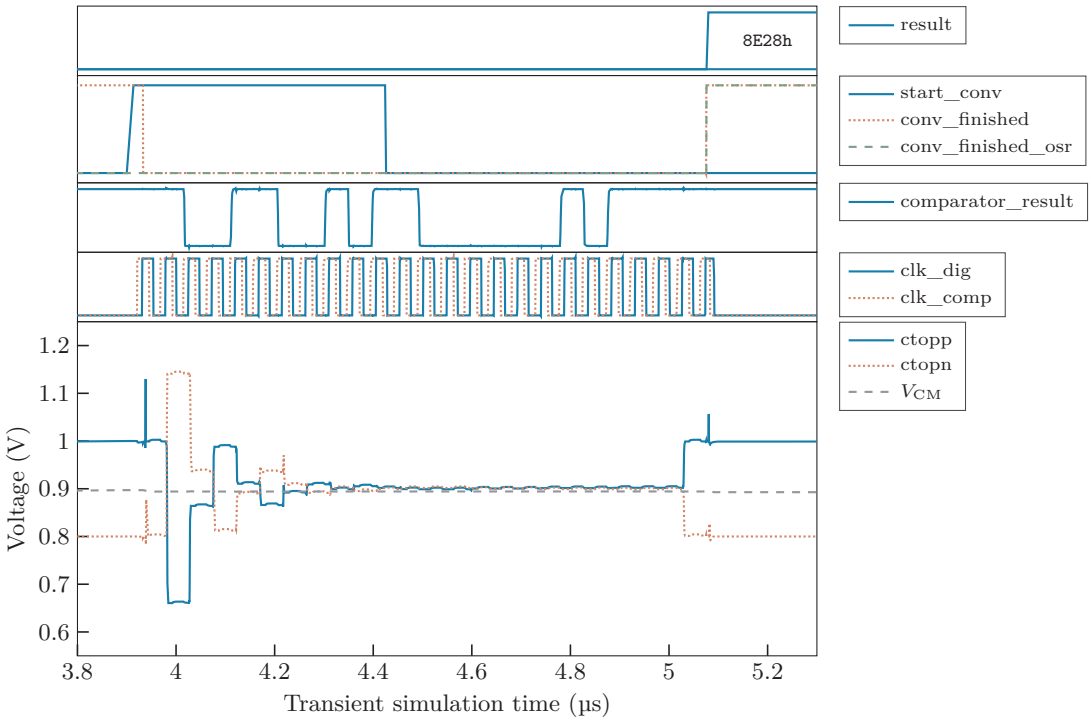



**Figure 5.11:** Post-layout simulation of a single conversion with averaging using 3 samples per weight, and oversampling of 4 conversions. The last conversion cycle of 4 conversion cycles is shown. The differential voltage $V_{\text{inp}} - V_{\text{inn}}$ has been set to 200 mV.

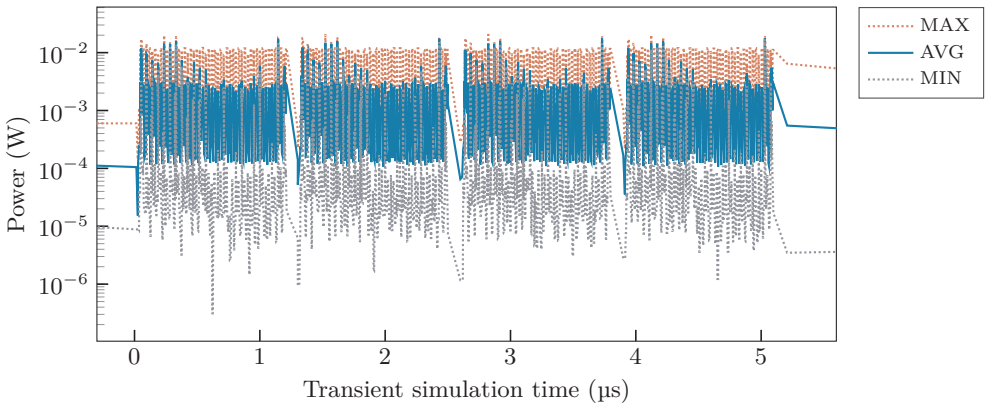



**Figure 5.12:** Power consumption of the SAR-ADC during the conversion shown in Fig. 5.10.

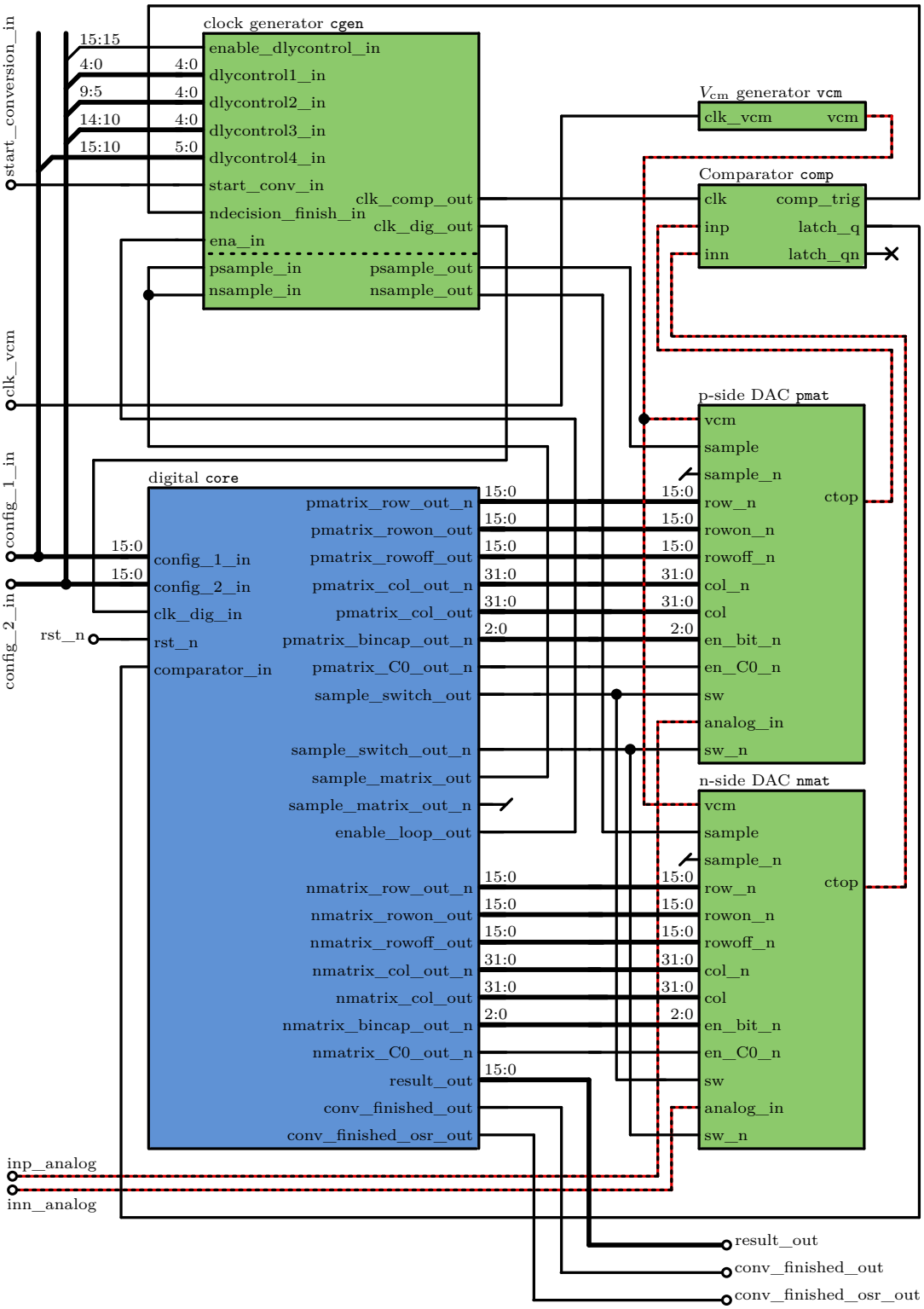



**Figure 5.13:** The block diagram of the top-level design. The blue block is the digital RTL logic, and the green blocks are the analog macro modules. Analog wires have been drawn as red and black dotted lines.

# Chapter 6

# Summary and Outlook

In the context of this Master's thesis, a versatile 1.2 MS/s 12-bit SAR-ADC based on the SKY130 process node has been designed using an area of only 0.178 $mm^2$. The area reduces to 0.124 $mm^2$ without the $V_{CM}$ generator. The workflows for analog, digital, and mixed-signal designs have been limited to the use of only free and open-source tools, which makes this design publicly available.

A custom standard cell with a large signal delay for the design of time-dependent, asynchronous circuits, has been designed for the SKY130 process node. Furthermore, a method to use the automated workflow of `OpenLane` for the hardening of a GDSII layout using the custom standard cell in a gate-level HDL description, has been developed. This allows the designer to describe a time-dependent circuit with Verilog HDL while a digital workflow is used for the automated generation of the layout. As a result, layout changes of the clock generator are done with only a few lines of code.

The digital clock frequency of this generator is configurable in the range of $f_{clk\_dig}$ = 1.016 MHz to 20.96 MHz using digital input ports. The implemented averaging and oversampling functions can be enabled to increase the precision of the ADC in the least significant SAR weights, whereas also the sample rate can be further modified. The sampling rate can be altered in the range from 1203 kS/s to 0.03 kS/s while oversampling increases the resolution to 16 bit. The lowered sampling rate reduces the power consumption down to 68 µW which makes the ADC interesting for low-power biomedical sensor applications.

Thermometer-coded row- and column-decoders in the DAC allow the efficient implementation of a non-binary algorithm with an error correction capability of 435 LSB at the first comparator decision. The implementation of thermometer-code DAC control with sequential or symmetric row and column scan modes enables the designer to choose a compromise between DNL or INL error. At a typical rate of $f_s$ = 823.55 kS/s the SAR-ADC dissipates 334.97 µW in the simulation. The broad configuration capability permits the usage of this design in a wide variety of different use cases.

The simulation results of this work have been compared to other state-of-the-art designs in Fig. 6.1, whereas most of these designs are biomedical sensor applications.

**Table 6.1:** Performance comparison between state-of-the-art SAR ADC designs as seen in [9].

| | This work | [26] | [9] | [27] | [28] | [29] | [30] |
|---|---|---|---|---|---|---|---|
| Process node | SKY130 | SKY130 | 180 nm CMOS | 55 nm Fujitsu | 180 nm CMOS | 65 nm CMOS | 180 nm CMOS |
| $V_{DD}$ (V) | 1.8 | 1.8 | 1.0 | 0.5/0.9 | 1.5 | 1.0 | 1.2 |
| DAC (bit) | 12 | 10 | 14 | 12 | 12 | 9 | 10 |
| Area ($mm^2$) | 0.178[a] | 0.149 | 0.349 | 0.27 | 0.192 | 0.072 | 0.13 |
| OSR (1) | 1-256 | 1 | 4 | 1 | 8 | 8 | 1 |
| BW (kHz) | 0.014-602 | 780 | 0.512 | 500 | 320 | 625 | 0.2 |
| Power (µW) | 68[b]/335[c] | 2100 | 1.125 | 30 | 180.1 | 130 | 0.1 |
| SNDR (dB) | - | 57.8 | 78.58 | 68 | 74.8 | 70.98 | - |

[a] Includes switched-capacitor $V_{CM}$ generator.
[b] Low-power test case, $\text{delay}_{1,2,3}$ = `11111`, $N_{avg} = 31$, $f_s = 7.36\,kS/s$.
[c] Typical test case, $\text{delay}_{1,2,3}$ = `00001`, $N_{avg} = 3$, $f_s = 0.82\,MS/s$.

Overall the designs utilizing the SKY130 PDK show a higher current consumption than commercial process nodes, but it must be noted that the supply voltage is the highest of all designs in the comparison.

The design needs to be characterized after tape out to verify the simulation results with appropriate measurement equipment. SNDR has not been evaluated due to the high computing effort, a complete simulation with R and C parasitics would not have been practically feasible.

In future designs, the power consumption might be reduced, because it should be possible to switch the digital standard cell library from high-density to low-power `sky130_fd_sc_lp`. Additionally, it is possible to switch the synthesis strategy to an area-efficient mode if a decreased sampling speed can be tolerated, this is expected to reduce the gate count. In this work, a strategy focused on a tradeoff between speed and area has been chosen.

This work has shown what open-source chip design EDA tools are capable of today. Free and open-source tools can be serious competitors to industrial standards and PDKs for sub-micron analog, digital and mixed-signal chip design. Especially for educational purposes, universities, and self-study, the availability of PDKs and a holistic design environment free-of-charge enables everyone to learn chip design from specification to tape out, hands-on and without the need to pay for expensive licenses. As of July $28^{th}$ [25], Google announced the work on a 90nm fully-depleted silicon-on-insulator PDK in collaboration with SkyWater Technology. Only some weeks ago, a new IHP Open-Source PDK based on a 130 nm BiCMOS process has been released. These and other projects indicate the trend of rising interest in the development of open-source chip design toward more advanced process nodes.

# Appendix A

# Appendix

## A.1 Clock Generation Loop with `clkdlybuf4s50_2`

### A.1.1 Verilog HDL of `adc_clkgen_with_edgedetect_native`

**Listing A.1:** Hardware description of macro `adc_clkgen_with_edgedetect_native` using the available SKY130 standard cells as the delay element.

```
1  module adc_clkgen_with_edgedetect_native(ena_in,start_conv,comp_trig,clk_dig,clk_comp);
2    input ena_in,input start_conv,input comp_trig;
3    output clk_dig,output clk_comp;
4    wire start_edge_detect;
5
6    adc_edge_detect_circuit edgedetect (.start_conv(start_conv), .ena_in(ena_in),
     ↪  .ena_out(start_edge_detect));
7    adc_clk_generation clkgen (.comp_trig(comp_trig), .ena(start_edge_detect),
     ↪  .clk_dig(clk_dig), .clk_comp(clk_comp));
8  endmodule
9
10 module adc_edge_detect_circuit(start_conv,ena_in,ena_out);
11     input ena_in, start_conv;
12     output ena_out;
13     wire start_conv_delayed, wire start_conv_edge;
14
15     delaycell #(.Ntimes(540)) delay_200ns (.in(start_conv), .out(start_conv_delayed));
16     nor(start_conv_edge,~start_conv,start_conv_delayed);
17     or(ena_out,start_conv_edge,ena_in);
18 endmodule
19
20 module adc_clk_generation(comp_trig,ena,clk_dig,clk_comp);
21    input comp_trig, ena;
22    output clk_dig, clk_comp;
23    wire comp_trig_delayed, ncomp_trig_delayed, net_1;
24
25    delaycell #(.Ntimes(270)) delay_100ns_1 (.in(comp_trig), .out(comp_trig_delayed));
26    assign clk_dig = ~comp_trig_delayed; //needs a buffer?
27    delaycell #(.Ntimes(270)) delay_100ns_2 (.in(~comp_trig_delayed),
     ↪  .out(ncomp_trig_delayed));
28    nor(net_1,ncomp_trig_delayed,~ena);
29    delaycell #(.Ntimes(270)) delay_100ns_3 (.in(net_1), .out(clk_comp));
30 endmodule
31
32 module delaycell #(parameter N_TIMES = 32)
33 (
34     input wire in,
35     output wire out
```

```
);
    wire [N_TIMES:0] signal_w;
    genvar j;
    generate
       for(j=0;j<N_TIMES;j=j+1) begin
       sky130_fd_sc_hd__clkdlybuf4s50_2 delay_unit (.in(signal_w[j]), .out(signal_w[j+1]));
       end
    endgenerate
    assign in = signal_w[0];
    assign out = signal_w[N_TIMES];
endmodule
```

### A.1.2 OpenLane configuration for adc_clkgen_with_edgedetect_native

**Listing A.2:** OpenLane configuration file config.tcl for hardening of the clock generator and edge detection macro adc_clkgen_with_edgedetect_native.gds using available SKY130 digital standard cells.

```
set ::env(DESIGN_NAME) adc_clkgen_with_edgedetect_native
set ::env(CLOCK_PORT) ""
set ::env(CLOCK_NET) $::env(CLOCK_PORT)
set ::env(CLOCK_TREE_SYNTH) 0

set ::env(VERILOG_FILES) [glob $::env(DESIGN_DIR)/src/*.v]
set ::env(SYNTH_READ_BLACKBOX_LIB) 1
set ::env(FP_SIZING) "relative"
set ::env(FP_ASPECT_RATIO) {1}
set ::env(FP_CORE_UTIL) {50}

set ::env(FP_PDN_HOFFSET) 20
set ::env(FP_PDN_VOFFSET) $::env(FP_PDN_HOFFSET)
set ::env(FP_PDN_HPITCH) 100
set ::env(FP_PDN_VPITCH) $::env(FP_PDN_HPITCH)
set ::env(FP_PIN_ORDER_CFG) $::env(DESIGN_DIR)/pin_order.cfg
set ::env(DESIGN_IS_CORE) 0
set ::env(FP_PDN_CORE_RING) 0
set ::env(RT_MAX_LAYER) {met4}
set ::env(VDD_NETS) [list {VPWR} {VPB}]
set ::env(GND_NETS) [list {VGND} {VNB}]
set ::env(PL_BASIC_PLACEMENT) 0
set ::env(PL_TARGET_DENSITY) {0.90}
set ::env(PL_RESIZER_TIMING_OPTIMIZATIONS) {0}
set ::env(PL_RESIZER_DESIGN_OPTIMIZATIONS) {0}
set ::env(PL_ROUTABILITY_DRIVEN) {1}
set ::env(GLB_RESIZER_TIMING_OPTIMIZATIONS) {0}

set filename $::env(DESIGN_DIR)/$::env(PDK)_$::env(STD_CELL_LIBRARY)_config.tcl
if { [file exists $filename] == 1} {
    source $filename
}
```

## A.2 Configurations for Standard Cells in magic

**Listing A.3:** Standard cell configuration settings in magic

LEF class properties for custom standard cells:

```
property LEFclass CORE
property FIXED_BBOX {AX AY BX BY}
```

```
property LEForigin {0 0}
property LEFsite unithd
```

The settings for the power port VPWR and substrate connection VPB:

```
port use power
port class inout
port shape abutment
```

Settings for the ground port VGND and substrate connection VNB:

```
port use ground
port class inout
port shape abutment
```

Settings for the input signal ports:

```
port use signal
port class input
```

Settings for the output signal ports:

```
port use signal
port class output
```

## A.3 Clock Generator with sky130_mm_sc_hd_dlyPoly5ns

### A.3.1 Verilog gate level HDL of macro adc_clkgen_with_edgedetect

**Listing A.4:** Verilog gate level HDL of macro adc_clkgen_with_edgedetect using the custom delay standard cell as delay element.

```
`default_nettype none
//  Copyright 2022 Manuel Moser
//
//   Licensed under the Apache License, Version 2.0 (the "License");
//   you may not use this file except in compliance with the License.
//   You may obtain a copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in writing, software
//   distributed under the License is distributed on an "AS IS" BASIS,
//   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
//   See the License for the specific language governing permissions and
//   limitations under the License.


// Asynchronous Clockgenerator with Edge-Detection for self-clocked ADC.
//
// Important: This verilog-file can not be synthesized with openlane-Optimizations
//            activated. Delays will be subsitituted otherwise. Verify in xschem with
↪  postlayout extraction
//            Harden with
↪  https://github.com/w32agobot/SKY130-RTL-with-Custom-Standardcell-to-GDSII
```

```
22
23  module adc_clkgen_with_edgedetect(
24      input wire ena_in,              // enable signal from the digital clock core. 0 halts the
        ↪  self-clocked loop
25      input wire start_conv_in,         // triggers a conversion once with edge-detection
26      input wire ndecision_finish_in,   // comparator signalizes finished conversion
27      output wire clk_dig_out,           // digital clock
28      output wire clk_comp_out,          // comparator clock
29      input wire enable_dlycontrol_in,  // 0 = max delays, 1 = configurable delays
30      input wire [4:0] dlycontrol1_in,  // delay 1 of 3 in loop. Delay = 5ns*dlycontrol1
31      input wire [4:0] dlycontrol2_in,  // delay 2 of 3 in loop. Delay = 5ns*dlycontrol2
32      input wire [4:0] dlycontrol3_in,  // delay 3 of 3 in loop. Delay = 5ns*dlycontrol3
33      input wire [5:0] dlycontrol4_in,  // edge detect pulse width. Delay = 5ns*dlycontrol4
34      // integration of additional buffers for sample matrix
35      input wire sample_p_in,
36      input wire sample_n_in,
37      output wire sample_p_out,
38      output wire sample_n_out
39  );
40     wire enable_loop_w;
41     wire ena_in_buffered_w;
42     wire start_conv_buffered_w;
43     wire ndecision_finish_buffered_w;
44     wire clk_dig_unbuffered_w;
45     wire clk_comp_unbuffered_w;
46
47     //Input buffers
48     sky130_fd_sc_hd__buf_1 inbuf_1 (.A(ena_in),.X(ena_in_buffered_w));
49     sky130_fd_sc_hd__buf_1 inbuf_2 (.A(start_conv_in),.X(start_conv_buffered_w));
50     sky130_fd_sc_hd__buf_1 inbuf_3 (.A(ndecision_finish_in),.X(ndecision_finish_buffered_w));
51
52     //Output buffers
53     sky130_fd_sc_hd__bufbuf_8 outbuf_1 (.A(clk_dig_unbuffered_w),.X(clk_dig_out));
54     sky130_fd_sc_hd__bufbuf_8 outbuf_2 (.A(clk_comp_unbuffered_w),.X(clk_comp_out));
55
56     // Output buffers for sample-signal-buffeing
57     // with delay, so matrix-sample is switched after gate is disabled
58     wire sample_p_1;
59     wire sample_n_1;
60
61     //sample enable delay stage
62     sky130_mm_sc_hd_dlyPoly5ns delay_sample_p11 (.in(sample_p_in), .out(sample_p_1));
63     sky130_mm_sc_hd_dlyPoly5ns delay_sample_n12 (.in(sample_n_in), .out(sample_n_1));
64
65     //sample enable output stage
66     sky130_fd_sc_hd__bufbuf_8 outbuf_3 (.A(sample_p_1),.X(sample_p_out));
67     sky130_fd_sc_hd__bufbuf_8 outbuf_4 (.A(sample_n_1),.X(sample_n_out));
68
69
70
71     //Circuits
72     adc_edge_detect_circuit edgedetect (.start_conv_in(start_conv_buffered_w),
73                                         .ena_in(ena_in_buffered_w),
74                                         .ena_out(enable_loop_w),
75                                         .enable_dlycontrol_in(enable_dlycontrol_in),
76                                         .dlycontrol_in(dlycontrol4_in));
77
78     adc_clk_generation clkgen (.ndecision_finish_in(ndecision_finish_buffered_w),
79                                .enable_loop_in(enable_loop_w),
80                                .clk_dig_out(clk_dig_unbuffered_w),
81                                .clk_comp_out(clk_comp_unbuffered_w),
82                                .enable_dlycontrol_in(enable_dlycontrol_in),
83                                .dlycontrol1_in(dlycontrol1_in),
```

```
84                              .dlycontrol2_in(dlycontrol2_in),
85                              .dlycontrol3_in(dlycontrol3_in));
86  endmodule
87
88  // generate the clock signals for comparator and digital core of the self-clocked
    ↪  SKY130_12Bit-SAR-ADC.
89  module adc_clk_generation(
90     input wire ndecision_finish_in,   // 0 = comparator finished
91     input wire enable_loop_in,        // 1 = self clocked loop enabled
92     output wire clk_dig_out,           // clk for digital core
93     output wire clk_comp_out,          // clk for comparator
94     input wire enable_dlycontrol_in,  // 0 = max delays, 1 = configured delays
95     input wire [4:0] dlycontrol1_in,  // delay1 = N times 5ns
96     input wire [4:0] dlycontrol2_in,  // delay2 = N times 5ns
97     input wire [4:0] dlycontrol3_in   // delay3 = N times 5ns
98  );
99     wire ndecision_finish_delayed_w;
100    wire clk_dig_delayed_w;
101    wire net1_w;
102
103     binary_delaycell #(.DLYCONTROL_BITWIDTH(5)) delay_155ns_1
104     (
105         .in(ndecision_finish_in),
106         .enable_dlycontrol_in(enable_dlycontrol_in),
107         .dlycontrol_in(dlycontrol1_in),
108         .out(ndecision_finish_delayed_w)
109     );
110     sky130_fd_sc_hd__inv_2 clkdig_inverter (.A(ndecision_finish_delayed_w),.Y(clk_dig_out));
111     binary_delaycell #(.DLYCONTROL_BITWIDTH(5)) delay_155ns_2
112     (
113         .in(clk_dig_out),
114         .enable_dlycontrol_in(enable_dlycontrol_in),
115         .dlycontrol_in(dlycontrol2_in),
116         .out(clk_dig_delayed_w)
117     );
118     sky130_fd_sc_hd__nor2b_1 nor1 (.A(clk_dig_delayed_w),.B_N(enable_loop_in),.Y(net1_w));
    ↪  //2 input nor, B inverted
119     binary_delaycell #(.DLYCONTROL_BITWIDTH(5)) delay_155ns_3
120     (
121         .in(net1_w),
122         .enable_dlycontrol_in(enable_dlycontrol_in),
123         .dlycontrol_in(dlycontrol3_in),
124         .out(clk_comp_out)
125     );
126 endmodule
127
128 // if ena_in is HIGH, then ena_out is HIGH.
129 // Else, detect the positive edge of start_conv and generate a pulsed signal at ena_out
    ↪  (low->high->low)
130 module adc_edge_detect_circuit(
131     input wire start_conv_in,          // Tell the ADC to start a conversion
132     input wire ena_in,              // signal from the digital core to enable the
    ↪  self-clocked-loop
133     output wire ena_out,            // enable the self-clocked-loop
134     input wire enable_dlycontrol_in,   // 0 = max delays, 1 = configured delays
135     input wire [5:0] dlycontrol_in     // delay = N times 5ns
136 );
137    // Internal wires
138     wire start_conv_delayed_w;
139     wire start_conv_edge_w;
140
141     binary_delaycell #(.DLYCONTROL_BITWIDTH(6)) dly_315ns_1
142     (
```

```
143         .in(start_conv_in),
144         .enable_dlycontrol_in(enable_dlycontrol_in),
145         .dlycontrol_in(dlycontrol_in),
146         .out(start_conv_delayed_w)
147     );
148     sky130_fd_sc_hd__nor2b_1 nor1
        ↪  (.A(start_conv_delayed_w),.B_N(start_conv_in),.Y(start_conv_edge_w)); // 2 input nor,
        ↪  B inverted
149     sky130_fd_sc_hd__or2_1 or1 (.A(start_conv_edge_w),.B(ena_in),.X(ena_out)); // 2 input or
150 endmodule
151
152
153 // ##################################################
154 // delay cell with bypass control and enable-pin
155 // Delays eare binary coded. control_bitwidth sets the number
156 // of delay cells. Number of cells is 2^(control_bitwidth)-1
157 // Maximum delay is 5ns*(2^control_bitwidth-1)
158 module binary_delaycell #(parameter DLYCONTROL_BITWIDTH = 5)
159 (
160     input wire in,
161     input wire enable_dlycontrol_in,
162     input wire [DLYCONTROL_BITWIDTH-1:0] dlycontrol_in,
163     output wire out
164 );
165
166     wire [DLYCONTROL_BITWIDTH:0] signal_w;
167     wire enable_dlycontrol_w;
168     wire [DLYCONTROL_BITWIDTH-1:0] bypass_enable_w;
169     wire [DLYCONTROL_BITWIDTH-1:0] bypass_w;
170
171     sky130_fd_sc_hd__buf_4 enablebuffer (.A(enable_dlycontrol_in),.X(enable_dlycontrol_w));
172
173     //instanciate binary coded delay cells
174     genvar i;
175     generate
176        for (i = 0; i < DLYCONTROL_BITWIDTH; i=i+1) begin
177             sky130_fd_sc_hd__inv_2  control_invert
             ↪  (.A(dlycontrol_in[i]),.Y(bypass_enable_w[i]));
178             sky130_fd_sc_hd__and2_1 bypass_enable
             ↪  (.A(enable_dlycontrol_w),.B(bypass_enable_w[i]),.X(bypass_w[i])); // 2 input
             ↪  and, A inverted
179             delaycell #(.N_TIMES_5NS(2**i)) dly_binary (
180                 .in(signal_w[i]),
181                 .bypass_in(bypass_w[i]),
182                 .out(signal_w[i+1])
183             );
184         end
185     endgenerate
186     assign signal_w[0] = in;
187     assign out = signal_w[DLYCONTROL_BITWIDTH];
188 endmodule
189
190
191     // delaycell with bypass function
192     //                                 ___________                 bypass
193     //             _____           |   Ntimes  |                 |-------\
194     // bypass-----o| AND |--siga[0]--| Delay 5ns |------siga[N]---|0 mux   |--out--
195     //       in-.--|_____|           |___________|   .--in--------|1______/
196     //           \_____________________________________/
197     //
198 module delaycell #(parameter N_TIMES_5NS = 32)
199 (
200     input wire in,
```

```
    input wire bypass_in,
    output wire out
);
    wire [N_TIMES_5NS:0] signal_w;
    genvar j;
    generate
       for(j=0;j<N_TIMES_5NS;j=j+1) begin
          sky130_mm_sc_hd_dlyPoly5ns delay_unit (.in(signal_w[j]), .out(signal_w[j+1]));
       end
    endgenerate
    sky130_fd_sc_hd__and2b_1 and_bypass_switch (.A_N(bypass_in),.B(in),.X(signal_w[0])); // 2 input and, A inverted
    sky130_fd_sc_hd__mux2_1 out_mux (.A0(signal_w[N_TIMES_5NS]),.A1(in),.S(bypass_in),.X(out)); //2 input mux
endmodule
```

### A.3.2 OpenLane configuration for macro adc_clkgen_with_edgedetect

**Listing A.5:** OpenLane configuration file config.tcl for hardening of the clock generator and edge detection layout adc_clkgen_with_edgedetect.gds using custom standard cells.

```
set ::env(DESIGN_NAME) adc_clkgen_with_edgedetect
# technically unused because combinatoric. Values are workarounds.
# otherwise the flow will fail with STA errors, or because the
# STA report will not be found although STA is skipped intentionally
 set ::env(CLOCK_PORT) ""
 set ::env(CLOCK_NET) ""
 set ::env(CLOCK_TREE_SYNTH) 0
 set ::env(CLOCK_PERIOD) 1000000

# Files
 set ::env(VERILOG_FILES) [glob $::env(DESIGN_DIR)/../../verilog/gl/adc_clkgen_with_edgedetect.v]
 set ::env(EXTRA_LEFS) [glob $::env(DESIGN_DIR)/../../stdcells/sky130_mm_sc_hd_dlyPoly5ns/sky130_mm_sc_hd_dlyPoly5ns.lef]
 set ::env(EXTRA_LIBS) [glob $::env(DESIGN_DIR)/../../stdcells/sky130_mm_sc_hd_dlyPoly5ns/sky130_mm_sc_hd_dlyPoly5ns.lib]
 set ::env(EXTRA_GDS_FILES) [glob $::env(DESIGN_DIR)/../../stdcells/sky130_mm_sc_hd_dlyPoly5ns/sky130_mm_sc_hd_dlyPoly5ns.gds]
 set ::env(SYNTH_READ_BLACKBOX_LIB) 1

# Floorplanning
 set ::env(FP_SIZING) "absolute"
 set ::env(DIE_AREA) "0 0 171 60"
 set ::env(BOTTOM_MARGIN_MULT) 1
 set ::env(TOP_MARGIN_MULT) 1
 set ::env(LEFT_MARGIN_MULT) 12
 set ::env(RIGHT_MARGIN_MULT) 12
 set ::env(GRT_ADJUSTMENT) 0.2

 # -synth_explore is helpful here
 set ::env(SYNTH_STRATEGY) "DELAY 4"
 set ::env(SYNTH_MAX_FANOUT) 12

# Power distribution network settings
 set ::env(FP_PDN_LOWER_LAYER) {met2}
 set ::env(FP_PDN_UPPER_LAYER) {met3}
 set ::env(RT_MAX_LAYER) {met3}
 set ::env(VDD_NETS) "VDD"
 set ::env(GND_NETS) "VSS"
 set ::env(FP_PDN_HOFFSET) 20
 set ::env(FP_PDN_HPITCH) 20
 set ::env(FP_PDN_VOFFSET) 50
 set ::env(FP_PDN_VPITCH) 50
 set ::env(FP_PIN_ORDER_CFG) $::env(DESIGN_DIR)/pin_order.cfg

# PDN on Macro Level or Core Level
 set ::env(DESIGN_IS_CORE) 1
 set ::env(FP_PDN_CORE_RING) 0

# Placement
```

```
 set ::env(PL_BASIC_PLACEMENT) 0
 set ::env(PL_TARGET_DENSITY) {0.85}
 set ::env(PL_RESIZER_TIMING_OPTIMIZATIONS) {0}
 set ::env(PL_RESIZER_DESIGN_OPTIMIZATIONS) {0}
 set ::env(DIODE_INSERTION_STRATEGY) 4

# needed for Customcell DlyPoly6ns
 set ::env(FP_TAPCELL_DIST) 14.26


# Router
 set ::env(GLB_RESIZER_TIMING_OPTIMIZATIONS) {0}

# LVS
 set ::env(MAGIC_EXT_USE_GDS) {1}


set filename $::env(DESIGN_DIR)/$::env(PDK)_$::env(STD_CELL_LIBRARY)_config.tcl
if { [file exists $filename] == 1} {
  source $filename
}
```

## A.4 Digital core logic hardware description

### A.4.1 Verilog HDL of `adc_core_digital`

**Listing A.6:** Verilog HDL of `adc_core_digital`.

```
//  Copyright 2022 Manuel Moser
//
//   Licensed under the Apache License, Version 2.0 (the "License");
//   you may not use this file except in compliance with the License.
//   You may obtain a copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in writing, software
//   distributed under the License is distributed on an "AS IS" BASIS,
//   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
//   See the License for the specific language governing permissions and
//   limitations under the License.
`default_nettype none

// ********************************************
//   SAR-ADC Core Logic
// ********************************************

module adc_core_digital(
   input  wire rst_n,
   input  wire [15:0] config_1_in,
   input  wire [15:0] config_2_in,
   output wire [15:0] result_out,
   output wire conv_finished_out,
   output wire conv_finished_osr_out,
   // Connections to Comparator-Latch
   input  wire comparator_in,
   // Connections to Clockloop-Generator with Edgedetect
   input  wire clk_dig_in,
   output wire enable_loop_out,
   // Connections to Cap-Matrix
   output wire sample_matrix_out,
   output wire sample_matrix_out_n,
   output wire sample_switch_out,
   output wire sample_switch_out_n,
   output wire [31:0] pmatrix_col_out_n,
```

```
38      output wire [31:0] pmatrix_col_out,
39      output wire [15:0] pmatrix_row_out_n,
40      output wire [15:0] pmatrix_rowon_out_n,
41      output wire [15:0] pmatrix_rowoff_out_n,
42      output wire [2:0]  pmatrix_bincap_out_n,
43      output wire        pmatrix_c0_out_n,
44      output wire [31:0] nmatrix_col_out_n,
45      output wire [31:0] nmatrix_col_out,
46      output wire [15:0] nmatrix_row_out_n,
47      output wire [15:0] nmatrix_rowon_out_n,
48      output wire [15:0] nmatrix_rowoff_out_n,
49      output wire [2:0]  nmatrix_bincap_out_n,
50      output wire        nmatrix_c0_out_n
51  );
52
53  //Configuration bytes mapping
54  wire [2:0] avg_control_w = config_1_in[2:0];
55  wire [2:0] osr_mode_w = config_1_in[5:3];
56  wire row_mode_w = config_1_in[6];
57  wire col_mode_w = config_1_in[7];
58
59  // Sample switch enable
60  assign sample_switch_out  = sample_cnb;
61  assign sample_switch_out_n = sample_cnb_n;
62
63  // Matrix sampling enable
64  assign sample_matrix_out  = sample_cnb;
65  assign sample_matrix_out_n = sample_cnb_n;
66
67  //*****************************************
68  //      ADC Nonbinary Control-Block
69  //*****************************************
70  wire [11:0] result_cnb;
71  wire [11:0] pswitch_cnb;
72  wire [11:0] nswitch_cnb;
73  wire conv_finished_cnb_n;
74  wire sample_cnb_n;
75  wire sample_cnb;
76
77  adc_control_nonbinary cnb (
78     .clk(clk_dig_in),
79     .rst_n(rst_n),
80     .comparator_in(comparator_in),
81     .avg_control_in(avg_control_w),
82     .sample_out(sample_cnb),
83     .sample_out_n(sample_cnb_n),
84     .enable_loop_out(enable_loop_out),
85     .conv_finished_strobe_out(conv_finished_cnb_n),
86     .pswitch_out(pswitch_cnb),
87     .nswitch_out(nswitch_cnb),
88     .result_out(result_cnb)
89  );
90
91  assign conv_finished_out = conv_finished_cnb_n;
92
93  //*****************************************
94  //           P/N-Matrix decoder
95  //*****************************************
96  adc_row_col_decoder pdc (
97     .data_in(pswitch_cnb),
98     .row_mode(row_mode_w),
99     .col_mode(col_mode_w),
100    .row_out_n(pmatrix_row_out_n),
```

```
   .rowon_out_n(pmatrix_rowon_out_n),
   .rowoff_out_n(pmatrix_rowoff_out_n),
   .col_out_n(pmatrix_col_out_n),
   .col_out(pmatrix_col_out),
   .bincap_out_n(pmatrix_bincap_out_n),
   .c0p_out_n(pmatrix_c0_out_n),
   .c0n_out_n(_unused_ok_pin1)
);

adc_row_col_decoder ndc (
   .data_in(nswitch_cnb),
   .row_mode(row_mode_w),
   .col_mode(col_mode_w),
   .row_out_n(nmatrix_row_out_n),
   .rowon_out_n(nmatrix_rowon_out_n),
   .rowoff_out_n(nmatrix_rowoff_out_n),
   .col_out_n(nmatrix_col_out_n),
   .col_out(nmatrix_col_out),
   .bincap_out_n(nmatrix_bincap_out_n),
   .c0p_out_n(_unused_ok_pin2),
   .c0n_out_n(nmatrix_c0_out_n)
);

//*****************************************
//           Oversampling unit
//*****************************************
adc_osr osr (
   .rst_n(rst_n),
   .data_valid_strobe(conv_finished_cnb_n),
   .osr_mode_in(osr_mode_w),
   .data_in(result_cnb),
   .data_out(result_out),
   .conversion_finished_osr_out(conv_finished_osr_out)
);

//Linting
wire [7:0]  _unused_ok_1 = config_1_in[15:8];
wire [15:0] _unused_ok_2 = config_2_in[15:0];
wire _unused_ok_pin1;
wire _unused_ok_pin2;

endmodule
```

### A.4.2 Verilog HDL of `adc_control_nonbinary`

**Listing A.7:** Verilog HDL of `adc_control_nonbinary`.

```

//  Copyright 2022 Manuel Moser
//
//   Licensed under the Apache License, Version 2.0 (the "License");
//   you may not use this file except in compliance with the License.
//   You may obtain a copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in writing, software
//   distributed under the License is distributed on an "AS IS" BASIS,
//   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
//   See the License for the specific language governing permissions and
//   limitations under the License.
`default_nettype none

```

```
17
18  // Top module ADC Control
19  module adc_control_nonbinary #(parameter MATRIX_BITS = 12, NONBINARY_REDUNDANCY = 3)(
20     input wire clk,
21     input wire rst_n,
22     input wire comparator_in,
23     input wire [2:0] avg_control_in,
24     output wire sample_out,
25     output wire sample_out_n,
26     output wire enable_loop_out,
27     output wire conv_finished_strobe_out,
28     output wire[MATRIX_BITS-1:0]  pswitch_out,
29     output wire[MATRIX_BITS-1:0]  nswitch_out,
30     output reg[MATRIX_BITS-1:0]   result_out
31     );
32
33     // combinatoric signals for next register values
34     wire [MATRIX_BITS-1:0] next_result_w;
35     wire [4:0] next_average_counter_w;
36     wire [4:0] next_average_sum_w;
37     wire [2:0] next_sampled_avg_control_w;
38     wire [MATRIX_BITS+NONBINARY_REDUNDANCY+1:0] next_shift_register_w;
39     wire [MATRIX_BITS-1:0] next_data_register_w;
40
41     // Average calculation of comparator_in at LSB-region
42     reg [4:0] average_counter_r;
43     reg [4:0] average_sum_r;
44     reg [2:0] sampled_avg_control_r;
45     wire [4:0] average_count_limit_w;
46     wire averaged_comparator_in_w;
47     wire lsb_region_w        = (shift_register_r[2] | shift_register_r[3] |
       ↪  shift_register_r[4] | shift_register_r[5]);
48     wire is_averaging_w      = (lsb_region_w && (average_counter_r < average_count_limit_w));
49
50     // State-Machine Shift Register
51     reg [MATRIX_BITS+NONBINARY_REDUNDANCY+1:0] shift_register_r;
52     reg [MATRIX_BITS-1:0] data_register_r;
53     wire [MATRIX_BITS-1:0] nonbinary_value_w;
54
55    // State machine states
56     wire is_holding_result_w = shift_register_r[1];
57     wire is_sampling_w       = shift_register_r[0];
58     wire result_ready_w = ((shift_register_r[2]==1'b1)&~is_averaging_w);
59     wire result_strobe_w = ((shift_register_r[1]==1'b1)&~is_averaging_w);
60
61
62    /*
63    ************************************************
64    //wire hold_data_for_osr   = shift_register_r[1];
65    ************************************************
66
67      OSR uses the data-valid strobe as clock-signal for low power,
68      which can lead to unpredictable problems.
69
70      expected:
71      data_in         XXXXX__data___XXXX
72      data_valid      _____/°°°°°°\_____
73      data_valid_clk  ____________/°°\__
74
75      Possible problem:
76      data_in         XXXXX__data___XXXXXX
77      data_valid      _____/°°°°°°\_______
78      data_valid_clk  ______________/°°\__ Clock is delayed
```

```
79
80      Solution: Data at OSR input is held for two clock cycles
81      data_in         XXXXX__data_________XXXXXX
82      data_valid      _____/°°°°°°\_______
83      data_valid_clk  ______________/°°\__ Clock can have delay
84
85    */
86
87
88    // conversion finished set 0 after reset, 1 after conversion ended
89    wire next_conv_finished_w = result_strobe_w;
90    reg  conv_finished_r;
91
92
93    //*****************************************
94    //   Synchronous process and Reset Handling
95    //*****************************************
96    always @(posedge clk, negedge rst_n) begin
97       if (rst_n == 1'b0) begin
98          result_out            <= {MATRIX_BITS{1'b0}};
99          shift_register_r      <= {{(MATRIX_BITS+NONBINARY_REDUNDANCY+1){1'b0}},1'b1};
100         sampled_avg_control_r <= 3'b000;
101         average_counter_r     <= 5'd1;
102         average_sum_r         <= 5'd0;
103         data_register_r       <= 12'd2048;
104         conv_finished_r       <= 1'b0;
105      end
106      else begin
107         result_out            <= next_result_w;
108         shift_register_r      <= next_shift_register_w;
109         sampled_avg_control_r <= next_sampled_avg_control_w;
110         average_counter_r     <= next_average_counter_w;
111         average_sum_r         <= next_average_sum_w;
112         data_register_r       <= next_data_register_w;
113         conv_finished_r       <= next_conv_finished_w;
114      end
115   end
116
117   //*******************************
118   //   Combinatorial Processes
119   //*******************************
120
121   // direct output values determined from internal registers
122   assign sample_out        = is_sampling_w;
123   assign sample_out_n      = ~is_sampling_w;
124   assign conv_finished_strobe_out = conv_finished_r;
125   assign enable_loop_out        = ~is_sampling_w;
126   assign pswitch_out      = ~data_register_r;
127   assign nswitch_out      = data_register_r;
128
129   // shift register and data handling
130   // save avg_control in a register to prevent changes of this value during conversion
131   assign next_shift_register_w      = is_averaging_w ? shift_register_r :
      ↪ {shift_register_r[0],shift_register_r[MATRIX_BITS+NONBINARY_REDUNDANCY+1:1]};
132   assign next_sampled_avg_control_w = is_sampling_w ? avg_control_in :
      ↪  sampled_avg_control_r;
133
134   wire [MATRIX_BITS-1:0] sar_up   = data_register_r+nonbinary_value_w;
135   wire [MATRIX_BITS-1:0] sar_down = data_register_r-nonbinary_value_w;
136   assign next_data_register_w = is_sampling_w | is_holding_result_w | result_ready_w ?
      ↪  12'd2048 :
137                                 is_averaging_w ? data_register_r :
138                                 averaged_comparator_in_w ? sar_up : sar_down ;
```

```
139
140     // update the result at end of conversion
141     //assign next_result_w = result_ready_w ? next_data_register_w : result_out;
142     assign next_result_w = (result_ready_w &  averaged_comparator_in_w) ? data_register_r   :
143                            (result_ready_w & ~averaged_comparator_in_w) ? data_register_r-1 :
144                            result_out;
145
146     // Get the comparator_in average value.
147     // Sum up comparator_in while in LSB region.
148     // Afterwards the result in average_sum is evaluated.
149     assign next_average_counter_w   = is_averaging_w ? average_counter_r+1 : 5'd1;
150     assign next_average_sum_w       = is_averaging_w ? average_sum_r+{4'b0,comparator_in} :
        ↪ {4'b0, comparator_in};
151     assign averaged_comparator_in_w = (~lsb_region_w) ? comparator_in :
152                                       is_averaging_w ? 1'b0 :
153                                       (average_count_limit_w ==  5'd3) ? average_sum_r[1] :
154                                       (average_count_limit_w ==  5'd7) ? average_sum_r[2] :
155                                       (average_count_limit_w ==  5'd15) ? average_sum_r[3] :
156                                       (average_count_limit_w ==  5'd31) ? average_sum_r[4] :
157                                       comparator_in;
158
159
160
161     //*******************************
162     //   NONBINARY Lookup table
163     //*******************************
164     // calculated for 12 Bit Matrix + 3 redundant Bits
165      assign nonbinary_value_w = (shift_register_r==17'd2**16) ? 12'd806 :
166                                 (shift_register_r==17'd2**15) ? 12'd486 :
167                                 (shift_register_r==17'd2**14) ? 12'd295 :
168                                 (shift_register_r==17'd2**13) ? 12'd180 :
169                                 (shift_register_r==17'd2**12) ? 12'd110 :
170                                 (shift_register_r==17'd2**11) ? 12'd67 :
171                                 (shift_register_r==17'd2**10) ? 12'd41 :
172                                 (shift_register_r==17'd2**9 ) ? 12'd25 :
173                                 (shift_register_r==17'd2**8 ) ? 12'd15 :
174                                 (shift_register_r==17'd2**7 ) ? 12'd9 :
175                                 (shift_register_r==17'd2**6 ) ? 12'd6 :
176                                 (shift_register_r==17'd2**5 ) ? 12'd4 :
177                                 (shift_register_r==17'd2**4 ) ? 12'd2 :
178                                 (shift_register_r==17'd2**3 ) ? 12'd1 :
179                                 (shift_register_r==17'd2**2 ) ? 12'd0 :
180                                 (shift_register_r==17'd2**1 ) ? 12'd0 :
181                                 (shift_register_r==17'd2**0 ) ? 12'd0 :
182                                 12'dX; // default signal for illegal state
183
184      //*******************************
185      //   AVERAGING lookup table
186      //*******************************
187      // Amount of measurements to average comparator_in at LSB_region
188      assign average_count_limit_w = (sampled_avg_control_r == 3'b001) ? 5'd3  :
189                                     (sampled_avg_control_r == 3'b010) ? 5'd7  :
190                                     (sampled_avg_control_r == 3'b011) ? 5'd15 :
191                                     (sampled_avg_control_r == 3'b100) ? 5'd31 :
192                                     5'd1;
193
194 endmodule
195
196
197
198
199
```

### A.4.3 Verilog HDL of `adc_osr`

**Listing A.8:** Verilog HDL of `adc_osr`.

```
1  //  Copyright 2022 Manuel Moser
2  //
3  //   Licensed under the Apache License, Version 2.0 (the "License");
4  //   you may not use this file except in compliance with the License.
5  //   You may obtain a copy of the License at
6  //
7  //       http://www.apache.org/licenses/LICENSE-2.0
8  //
9  //   Unless required by applicable law or agreed to in writing, software
10 //   distributed under the License is distributed on an "AS IS" BASIS,
11 //   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 //   See the License for the specific language governing permissions and
13 //   limitations under the License.
14 `default_nettype none
15
16
17 /*
18    ADC Oversampler of [1,4,16,64,256] samples where 4**N samples equal +N Bit resolution
19    Input vector has 12 Bit, result has 16 Bit
20    data_in =  [ 12 Bit input  ] Range 12'h000  = -VCC ... 12'hFFF  = +VCC
21    data_out = [ 16 Bit result ] Range 16'h0000 = -VCC ... 16'hFFF0 = +VCC
22 */
23
24 module adc_osr (
25    input wire rst_n,
26    input wire data_valid_strobe,
27    input wire [2:0] osr_mode_in,
28    input wire [11:0] data_in,
29    output wire [15:0] data_out,
30    output wire conversion_finished_osr_out
31    );
32
33    // internal registers
34    reg  [19:0] result_r;
35    reg  [2:0]  osr_mode_r;
36    reg  [8:0]  sample_count_r;
37    reg  [15:0] output_r;
38    reg         data_valid_r;
39
40    // combinatoric signals for next register values
41    wire [19:0] next_result_w;
42    wire [2:0]  next_osr_mode_w;
43    wire [8:0]  next_sample_count_w;
44    wire [15:0] next_output_w;
45
46    // Direct signals
47    assign conversion_finished_osr_out = data_valid_r & data_valid_strobe;
48
49    //****************************************
50    //   data_valid_strobe as clock input
51    //****************************************
52    // Cave: Handle with care, normally you should
53    // only use one clk-signal to guarantee synchronous
54    // execution without problems
55
56    always @(posedge data_valid_strobe, negedge rst_n) begin
57       if (rst_n == 1'b0) begin
58          result_r          <= 20'd0;
59          osr_mode_r        <=  3'd0;
60          sample_count_r    <=  9'd1;
```

```
61          output_r          <=  16'd0;
62          data_valid_r      <=  1'b0;
63        end
64        else begin
65          result_r        <= next_result_w;
66          osr_mode_r      <= next_osr_mode_w;
67          sample_count_r  <= next_sample_count_w;
68          output_r        <= next_output_w;
69          data_valid_r    <= next_data_valid_w;
70        end
71      end
72
73      wire next_data_valid_w = is_last_sample;
74
75
76      //*******************************
77      //   State flags
78      //*******************************
79      wire bypass_oversampling = ~(osr_mode_in[0] | osr_mode_in[1] | osr_mode_in==3'b100);
80      wire is_first_sample = bypass_oversampling | (sample_count_r == 9'd1);
81      wire is_last_sample  = bypass_oversampling | ((sample_count_r ==
        ↪  osr_count_limit_w)&&(~is_first_sample));
82
83      //*******************************
84      //   Combinatoric signals
85      //*******************************
86      assign next_result_w        = is_first_sample ? {8'd0,data_in} :
87                                                      {8'd0,data_in} + result_r;
88
89      assign next_osr_mode_w      = is_first_sample ? osr_mode_in : osr_mode_r;
90      assign next_sample_count_w  = is_last_sample ? 1 : sample_count_r + 1;
91
92       //*********************************************
93       //   Output right-shifted result
94       //*********************************************
95       assign data_out = output_r;
96
97       assign next_output_w = bypass_oversampling    ? {next_result_w[11:0],4'b0} :
98                              ~is_last_sample        ? output_r                   :
99                              (osr_mode_r == 3'b001) ? {next_result_w[13:1],3'b0} :
100                             (osr_mode_r == 3'b010) ? {next_result_w[15:2],2'b0} :
101                             (osr_mode_r == 3'b011) ? {next_result_w[17:3],1'b0} :
102                             (osr_mode_r == 3'b100) ? {next_result_w[19:4]}      :
103                             16'bX;
104
105     //*********************************************
106     //   OSR mode lookup table
107     //   mode 001 = 4 samples   +1 Bit resolution
108     //   mode 010 = 16 samples  +2 Bit resolution
109     //   mode 011 = 64 samples  +3 Bit resolution
110     //   mode 100 = 256 samples +4 Bit resolution
111     //*********************************************
112     // Amount of oversampling samples (+N Bit SNR per N*4 samples)
113     wire [8:0] osr_count_limit_w = (osr_mode_r == 3'b001) ? 9'd4   :
114                                    (osr_mode_r == 3'b010) ? 9'd16  :
115                                    (osr_mode_r == 3'b011) ? 9'd64  :
116                                    (osr_mode_r == 3'b100) ? 9'd256 :
117                                    9'd1;
118 endmodule
119
120
121
```

### A.4.4 Verilog HDL of `adc_row_col_decoder`

**Listing A.9:** Verilog HDL of `adc_row_col_decoder`.

```
/*
  Copyright 2022 Manuel Moser

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
*/
`default_nettype none

/*
   Convert Binary to (inverted) Thermo-Code for a
   capacitor-matrix with 16 rows, 32 columns and 3 Binary-Cells.

   row_mode=0 .. start with row 1 and progress up.. up.. up..
   row_mode=1 .. start at the middle row and progress up.. down.. up.. down..

   col_mode=0 .. start with col 1 (even) or col 32 (odd)
   col_mode=1 .. start at the middle column
*/

module adc_row_col_decoder(
    input  wire[11:0] data_in,
    input  wire row_mode,
    input  wire col_mode,
    output wire[15:0] row_out_n,
    output wire[15:0] rowon_out_n,
    output wire[15:0] rowoff_out_n,
    output wire[31:0] col_out_n,
    output wire[31:0] col_out,
    output wire[2:0]  bincap_out_n,
    output wire       c0p_out_n,
    output wire       c0n_out_n
        );


genvar i;

//[               data                ]
//[11][10][9][8][7][6][5][4][3][2][1][0]
//[    row     ][    col      ][bincap ]
wire[2:0] bincap_w = data_in[2:0];
wire[4:0] col_intermediate_w = data_in[7:3];
wire[3:0] row_intermediate_w = data_in[11:8];

wire row_direction_RL = row_intermediate_w[0] ; // romode  = 0 .. even/odd row is left to
↪  right
                                                // rowmode = 1 .. lower half rows
                                                ↪  are left to right
```

```
55
56  // *****************************************
57  //   Row Decoder
58  //
59  //   Row Mode 0: from bot to top MSB[8 7 6 5 4 3 2 1]LSB
60  //   Row Mode 1: start at middle MSB[7 5 3 1 2 4 6 8]LSB
61  // *****************************************
62
63  /// Row Down-to-Up mode
64  wire[15:0] row_bottotop_n = ~(16'h0001<<row_intermediate_w);
65  wire[15:0] rowon_bottotop_n = (16'hFFFF<<row_intermediate_w);
66
67  /// Row Mid-to-Top/Bot mode
68  wire [15:0] row_midtoside_n;
69  wire [15:0] rowon_midtoside_n;
70  generate
71    for (i=0;i<8;i=i+1) begin
72      assign row_midtoside_n[8+i] = row_bottotop_n[2*i];
73      assign row_midtoside_n[7-i] = row_bottotop_n[2*i+1];
74      assign rowon_midtoside_n[8+i] = rowon_bottotop_n[2*i];
75      assign rowon_midtoside_n[7-i] = rowon_bottotop_n[2*i+1];
76    end
77  endgenerate
78
79  /// Row Out
80  assign row_out_n = row_mode ? row_midtoside_n : row_bottotop_n ;
81  assign rowon_out_n = row_mode ? rowon_midtoside_n : rowon_bottotop_n ;
82
83
84
85  // status
86  wire is_first_row = ~row_out_n[0];
87
88
89  // *****************************************
90  //   Column Decoder
91  //
92  //   Col Mode 0: Direction from side to side
93  //      even row: direction is left-to-right MSB[8 7 6 5 4 3 2 1]LSB
94  //               odd row:  direction is right-to-left MSB[7 5 3 1 2 4 6 8]LSB
95  //   Col Mode 1: Direction from middle to side MSB[8 6 4 2 1 3 5 7]LSB
96  // *****************************************
97
98  // Shift register
99  wire[32:0] col_shift  = (33'h1FFFFFFFE)<<col_intermediate_w;
100 wire[32:0] col_shift_inv;
101 generate
102   for (i=0;i<33;i=i+1) begin
103     assign col_shift_inv[i] = col_shift[32-i];
104   end
105 endgenerate
106
107 //-COLUMN START VALUE OF SHIFT REGISTER because cell {0,0} is a Dummy-
108 // row  col  | first row   other rows
109 // 0    0    | 10..00      10..00
110 // 0    1    | 00..00      10..00
111 // 1    0    | 00..00      00..00
112 // 1    1    | 00..00      00..00
113
114 wire zeroes = row_mode == 1'b1 | (row_mode==1'b0 & col_mode==1'b1 & is_first_row );
115
116 wire[31:0] col_even_n = zeroes ? col_shift[32:1] : col_shift[31:0];
117 wire[31:0] col_odd_n  = zeroes ? col_shift_inv[31:0] :
```

```
                                col_shift_inv[32:1] ;

/// Column-Side-to-Side mode
wire[31:0] col_sidetoside_n = row_direction_RL ? col_odd_n : col_even_n;

/// Column Middle-to-Side mode
wire[31:0] col_midtoside_n;
generate
  for (i=0;i<16;i=i+1) begin
    assign col_midtoside_n[16+i] = col_even_n[2*i];
    assign col_midtoside_n[15-i] = col_even_n[2*i+1];
  end
endgenerate
/// Column Out
assign col_out_n = col_mode ? col_midtoside_n : col_sidetoside_n;
assign col_out = ~col_out_n;


// ********************************************
//   Bincap decoder, C0 decoder, misc
// ********************************************
assign bincap_out_n = ~bincap_w;

// semi-differential wires
assign rowoff_out_n = ~(row_out_n&rowon_out_n);

// LSB capacitor C0 is always enabled or disabled
assign c0p_out_n = 1'b0;
assign c0n_out_n = 1'b1;

endmodule
```

## A.5 Top-Level Hardening

### A.5.1 Verilog HDL of `adc_top`

**Listing A.10:** Verilog HDL of `adc_top`.

```
//  Copyright 2022 Manuel Moser
//
//   Licensed under the Apache License, Version 2.0 (the "License");
//   you may not use this file except in compliance with the License.
//   You may obtain a copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in writing, software
//   distributed under the License is distributed on an "AS IS" BASIS,
//   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
//   See the License for the specific language governing permissions and
//   limitations under the License.
`default_nettype none

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// NOTE: use RSZ_DONT_TOUCH_RX
// Reason -> no buffers on analog nets
//
// NOTE 2: MANUALLY CONENCT VCM AFTER OPENLANE FLOW
//
```

```
22  // NOTE 3: MANUALLY CONNECT DIGITAL CLOCK AFTER OPENLANE FLOW
23  //
24  //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
25
26  //Top module ADC Control
27  module adc_top(
28     `ifdef USE_POWER_PINS
29        inout VDD,        // User area 1.8V supply
30        inout VSS,        // User area ground
31     `endif
32     input wire clk_vcm, // 32.768Hz VCM generation clock
33     input wire rst_n,   // reset
34     input wire inp_analog,     // P differential input
35     input wire inn_analog,     // N differential input
36     input wire start_conversion_in,
37     input wire [15:0] config_1_in,
38     input wire [15:0] config_2_in,
39     output wire [15:0] result_out,
40     output wire conversion_finished_out,
41     output wire conversion_finished_osr_out,
42     input wire clk_dig_dummy
43     );
44
45  //Configuration byte 1 mapping
46  // config_1_in[2:0] = Average control
47  // config_1_in[5:3] = Oversampling control
48  // config_1_in[9:6] = unused
49  wire [5:0] delay_edgedetect_w = config_1_in[15:10];
50
51  //_linting
52  (*keep*)
53  wire _linting_unused_input_pins = config_1_in[6] | config_1_in[7] | config_1_in[8] |
    ↪  config_1_in[9];
54
55  //Configuration byte 2 mapping
56  wire [4:0] delay_1_w = config_2_in[4:0];
57  wire [4:0] delay_2_w = config_2_in[9:5];
58  wire [4:0] delay_3_w = config_2_in[14:10];
59  wire delaycontrol_en_w = config_2_in[15];
60
61  //*****************************************
62  //      Digital Core
63  //*****************************************
64  (*keep*)
65  adc_core_digital core(
66     .rst_n(rst_n),
67     .config_1_in(config_1_in),
68     .config_2_in(config_2_in),
69     .result_out(result_out),
70     .conv_finished_out(conversion_finished_out),
71     .conv_finished_osr_out(conversion_finished_osr_out),
72     // Connections to Comparator-Latch
73     .comparator_in(result_comp),
74     // Connections to Clockloop-Generator with Edgedetect
75     .clk_dig_in(clk_dig_dummy),
76     .enable_loop_out(ena_loop_core),
77     // Connections to Cap-Matrix
78     .sample_matrix_out(sample_matrix_core),
79     .sample_matrix_out_n(sample_matrix_core_n),
80     .sample_switch_out(sample_switch_core),
81     .sample_switch_out_n(sample_switch_core_n),
82     .pmatrix_col_out_n(pmatrix_col_core_n),
83     .pmatrix_col_out(pmatrix_col_core),
```

```
84      .pmatrix_row_out_n(pmatrix_row_core_n),
85      .pmatrix_rowon_out_n(pmatrix_rowon_core_n),
86      .pmatrix_rowoff_out_n(pmatrix_rowoff_core_n),
87      .pmatrix_bincap_out_n(pmatrix_bincap_core_n),
88      .pmatrix_c0_out_n(pmatrix_c0_core_n),
89      .nmatrix_col_out_n(nmatrix_col_core_n),
90      .nmatrix_col_out(nmatrix_col_core),
91      .nmatrix_row_out_n(nmatrix_row_core_n),
92      .nmatrix_rowon_out_n(nmatrix_rowon_core_n),
93      .nmatrix_rowoff_out_n(nmatrix_rowoff_core_n),
94      .nmatrix_bincap_out_n(nmatrix_bincap_core_n),
95      .nmatrix_c0_out_n(nmatrix_c0_core_n)
96  );
97
98  wire sample_matrix_core, sample_matrix_core_n;
99  wire sample_switch_core, sample_switch_core_n;
100 wire [31:0] pmatrix_col_core_n, nmatrix_col_core_n;
101 wire [31:0] pmatrix_col_core, nmatrix_col_core;
102 wire [15:0] pmatrix_row_core_n, nmatrix_row_core_n;
103 wire [15:0] pmatrix_rowon_core_n, nmatrix_rowon_core_n;
104 wire [15:0] pmatrix_rowoff_core_n, nmatrix_rowoff_core_n;
105 wire [2:0]  pmatrix_bincap_core_n, nmatrix_bincap_core_n;
106 wire        pmatrix_c0_core_n, nmatrix_c0_core_n;
107 wire ena_loop_core;
108
109
110 //*****************************************
111 //      Clock Loop with Edge-Detection
112 //      **** HARDENED MACRO ****
113 //*****************************************
114 (*keep*)
115 adc_clkgen_with_edgedetect cgen (
116    `ifdef USE_POWER_PINS
117       .VDD(VDD),        // User area 1.8V supply
118       .VSS(VSS),        // User area ground
119    `endif
120    .ena_in(ena_loop_core),
121    .start_conv_in(start_conversion_in),
122    .ndecision_finish_in(decision_finish_comp_n),
123    .clk_dig_out(clk_dig_cgen),
124    .clk_comp_out(clk_comp_cgen),
125    .enable_dlycontrol_in(delaycontrol_en_w),
126    .dlycontrol1_in(delay_1_w),
127    .dlycontrol2_in(delay_2_w),
128    .dlycontrol3_in(delay_3_w),
129    .dlycontrol4_in(delay_edgedetect_w),
130    .sample_p_in(sample_matrix_core),
131    .sample_n_in(sample_matrix_core),
132    .sample_p_out(sample_pmatrix_cgen),
133    .sample_n_out(sample_nmatrix_cgen)
134    );
135
136 wire clk_dig_cgen;
137 wire clk_comp_cgen;
138 wire sample_pmatrix_cgen, sample_nmatrix_cgen;
139
140 //*****************************************
141 //      Matrix P-side
142 //      **** HARDENED MACRO ****
143 //*****************************************
144 (*keep*)
145 adc_array_matrix_12bit pmat (
146    `ifdef USE_POWER_PINS
```

```
147       .VDD(VDD),        // User area 1.8V supply
148       .VSS(VSS),        // User area ground
149    `endif
150    .sample(sample_pmatrix_cgen),
151    .sample_n(~sample_pmatrix_cgen),
152    .row_n(pmatrix_row_core_n_buffered),
153    .rowon_n(pmatrix_rowon_core_n_buffered),
154    .rowoff_n(pmatrix_rowoff_core_n),
155    .col_n(pmatrix_col_core_n_buffered),
156    .col(pmatrix_col_core),
157    .en_bit_n(pmatrix_bincap_core_n),
158    .en_C0_n(pmatrix_c0_core_n),
159    .sw(pmat_sample_switch_buffered),
160    .sw_n(pmat_sample_switch_n_buffered),
161    .analog_in(inp_analog),
162    .ctop(ctop_pmatrix_analog)
163    );
164 wire ctop_pmatrix_analog;
165 //Buffering, switch signal must be fast
166 wire pmat_sample_switch_buffered, pmat_sample_switch_n_buffered;
167 sky130_fd_sc_hd__buf_8 pmat_sample_buf
    ↪  (.A(sample_switch_core),.X(pmat_sample_switch_buffered));
168 sky130_fd_sc_hd__buf_8 pmat_sample_buf_n
    ↪  (.A(sample_switch_core_n),.X(pmat_sample_switch_n_buffered));
169
170 //*****************************************
171 //      Matrix N-side
172 //      **** HARDENED MACRO ****
173 //*****************************************
174 (*keep*)
175 adc_array_matrix_12bit nmat (
176    `ifdef USE_POWER_PINS
177       .VDD(VDD),        // User area 1.8V supply
178       .VSS(VSS),        // User area ground
179    `endif
180    .sample(sample_nmatrix_cgen),
181    .sample_n(~sample_nmatrix_cgen),
182    .row_n(nmatrix_row_core_n_buffered),
183    .rowon_n(nmatrix_rowon_core_n_buffered),
184    .rowoff_n(nmatrix_rowoff_core_n),
185    .col_n(nmatrix_col_core_n_buffered),
186    .col(nmatrix_col_core),
187    .en_bit_n(nmatrix_bincap_core_n),
188    .en_C0_n(nmatrix_c0_core_n),
189    .sw(nmat_sample_switch_buffered),
190    .sw_n(nmat_sample_switch_n_buffered),
191    .analog_in(inn_analog),
192    .ctop(ctop_nmatrix_analog)
193    );
194 wire ctop_nmatrix_analog;
195 //Buffering, switch signal must be fast
196 wire nmat_sample_switch_buffered, nmat_sample_switch_n_buffered;
197 sky130_fd_sc_hd__buf_8 nmat_sample_buf
    ↪  (.A(sample_switch_core),.X(nmat_sample_switch_buffered));
198 sky130_fd_sc_hd__buf_8 nmat_sample_buf_n
    ↪  (.A(sample_switch_core_n),.X(nmat_sample_switch_n_buffered));
199
200
201 //*****************************************
202 //      Matrix Buffering
203 //      **** HARDENED MACRO ****
204 //*****************************************
205 genvar i;
```

```
206 wire [31:0] nmatrix_col_core_n_buffered;
207 wire [31:0] pmatrix_col_core_n_buffered;
208 generate
209   for (i=0;i<32;i=i+1) begin
210     sky130_fd_sc_hd__buf_8 buf_n_coln
      ↪  (.A(nmatrix_col_core_n[i]),.X(nmatrix_col_core_n_buffered[i]));
211     sky130_fd_sc_hd__buf_8 buf_p_coln
      ↪  (.A(pmatrix_col_core_n[i]),.X(pmatrix_col_core_n_buffered[i]));
212   end
213 endgenerate
214
215 wire [15:0] nmatrix_row_core_n_buffered;
216 wire [15:0] nmatrix_rowon_core_n_buffered;
217 wire [15:0] pmatrix_row_core_n_buffered;
218 wire [15:0] pmatrix_rowon_core_n_buffered;
219 generate
220   for (i=0;i<16;i=i+1) begin
221     sky130_fd_sc_hd__buf_4 buf_n_rown
      ↪  (.A(nmatrix_row_core_n[i]),.X(nmatrix_row_core_n_buffered[i]));
222     sky130_fd_sc_hd__buf_4 buf_n_rowonn
      ↪  (.A(nmatrix_rowon_core_n[i]),.X(nmatrix_rowon_core_n_buffered[i]));
223     sky130_fd_sc_hd__buf_4 buf_p_rown
      ↪  (.A(pmatrix_row_core_n[i]),.X(pmatrix_row_core_n_buffered[i]));
224     sky130_fd_sc_hd__buf_4 buf_p_rowonn
      ↪  (.A(pmatrix_rowon_core_n[i]),.X(pmatrix_rowon_core_n_buffered[i]));
225   end
226 endgenerate
227
228 //*****************************************
229 //      Comparator latch
230 //      **** HARDENED MACRO ****
231 //*****************************************
232 (*keep*)
233 adc_comp_latch comp (
234    `ifdef USE_POWER_PINS
235       .VDD(VDD),        // User area 1.8V supply
236       .VSS(VSS),        // User area ground
237    `endif
238    .clk(clk_comp_cgen),
239    .inp(ctop_pmatrix_analog),
240    .inn(ctop_nmatrix_analog),
241    .comp_trig(decision_finish_comp_n),
242    .latch_qn(_linting_unused_ok),
243    .latch_q(result_comp)
244    );
245    wire decision_finish_comp_n;
246    wire result_comp;
247    wire _linting_unused_ok;
248
249 //*****************************************
250 //      VCM generator
251 //      **** HARDENED MACRO ****
252 //*****************************************
253 (*keep*)
254 adc_vcm_generator vcm (
255    `ifdef USE_POWER_PINS
256       .VDD(VDD),        // User area 1.8V supply
257       .VSS(VSS),        // User area ground
258    `endif
259    .clk(clk_vcm)
260 );
261
262 (*keep*)
```

```
scboundary obstruction1 ();
(*keep*)
scboundary obstruction2 ();

endmodule


```

**Listing A.11:** OpenLane configuration file `config.json` for hardening of the top level layout `adc_top.gds`.

```
{
  "PDK": "sky130A",
  "STD_CELL_LIBRARY": "sky130_fd_sc_hd",
  "DESIGN_NAME": "adc_top",
  "VERILOG_FILES": [
      "dir::../../verilog/rtl/adc_row_col_decoder.v",
      "dir::../../verilog/rtl/adc_control_nonbinary.v",
      "dir::../../verilog/rtl/adc_osr.v",
      "dir::../../verilog/rtl/adc_core_digital.v",
      "dir::../../verilog/gl/adc_top.v"
      ],
  "VERILOG_FILES_BLACKBOX": [
      "dir::macros/blackbox/adc_array_matrix_12bit.v",
      "dir::macros/blackbox/adc_clkgen_with_edgedetect.v",
      "dir::macros/blackbox/adc_comp_latch.v",
      "dir::macros/blackbox/adc_vcm_generator.v",
      "dir::macros/blackbox/scboundary.v"
      ],
  "EXTRA_LEFS": [
      "dir::../../lef/adc_array_matrix_12bit.lef",
      "dir::../../lef/adc_clkgen_with_edgedetect.lef",
      "dir::../../lef/adc_comp_latch.lef",
      "dir::../../lef/adc_vcm_generator.lef",
      "dir::../../lef/scboundary.lef"
      ],
  "EXTRA_GDS_FILES": [
      "dir::../../gds/adc_array_matrix_12bit.gds",
      "dir::../../gds/adc_clkgen_with_edgedetect.gds",
      "dir::../../gds/adc_comp_latch.gds",
      "dir::../../gds/adc_vcm_generator.gds",
      "dir::../../gds/scboundary.gds"
      ],
  "FP_PIN_ORDER_CFG": "dir::pin_order.cfg",
  "MACRO_PLACEMENT_CFG": "dir::macro_placement.cfg",
  "FP_ENDCAP_CELL":"sky130_fd_sc_hd__tapvpwrvgnd_1",
  "PDN_CFG": "dir::pdn_cfg.tcl",
  "CLOCK_TREE_SYNTH": 1,
  "CLOCK_PORT": "clk_dig_dummy",
  "CLOCK_NET": "ref::$CLOCK_PORT",
  "CLOCK_PERIOD": 20,
  "RSZ_DONT_TOUCH_RX": ["analog"],
  "SYNTH_STRATEGY": "AREA 3",
  "FP_SIZING": "absolute",
  "DIE_AREA": [0, 0, 423, 403],
  "FP_PDN_VOFFSET": 12,
  "FP_PDN_VPITCH": 24,
  "FP_PDN_HOFFSET": 12,
  "FP_PDN_HPITCH": 23,
  "FP_TAP_HORIZONTAL_HALO": 3,
  "FP_TAP_VERTICAL_HALO": 3,
  "FP_PDN_HORIZONTAL_HALO": 3,
```

```
  "FP_PDN_VERTICAL_HALO": 3,
  "GRT_ADJUSTMENT": 0.15,
  "GRT_OBS": [
      "met1 287 0 302 130, ",
      "met2 287 0 302 130, ",
      "met3 287 0 302 130, ",
      "met4 287 0 302 130, ",
      "met1 287 270 302 391, ",
      "met2 287 270 302 391, ",
      "met3 287 270 302 391, ",
      "met4 287 270 302 391, ",
      "met1 302 229 404 391, ",
      "met2 302 229 404 391, ",
      "met3 302 229 404 391, ",
      "met4 302 229 404 391, ",
      "met1 302 9 404 171, ",
      "met2 302 9 404 171, ",
      "met3 302 9 404 171, ",
      "met4 302 9 404 171"
  ],
  "SYNTH_READ_BLACKBOX_LIB": true,
  "SYNTH_MAX_FANOUT": 6,
  "CLOCK_BUFFER_FANOUT": 16,
  "DPL_CELL_PADDING": 0,
  "DIODE_INSERTION_STRATEGY": 4,
  "DESIGN_IS_CORE": 1,
  "FP_PDN_CORE_RING": 1,
  "RT_MAX_LAYER": "met4",
  "VDD_NETS": ["VDD"],
  "GND_NETS": ["VSS"],
  "SYNTH_USE_PG_PINS_DEFINES": "USE_POWER_PINS",
  "PL_BASIC_PLACEMENT": 0,
  "PL_TARGET_DENSITY": 0.44,
  "PL_ROUTABILITY_DRIVEN": 1,
  "PL_TIME_DRIVEN": 0,
  "GRT_ALLOW_CONGESTION": 1,
  "ROUTING_CORES": 12,
  "MAGIC_EXT_USE_GDS": 1,
  "RUN_LVS": true,
  "FP_PDN_MACRO_HOOKS": [
      "vcm  VDD VSS VDD VSS, ",
      "pmat VDD VSS VDD VSS, ",
      "nmat VDD VSS VDD VSS, ",
      "comp VDD VSS VDD VSS, ",
      "cgen VDD VSS VDD VSS, ",
      "obstruction1 VDD VSS VDD VSS, ",
      "obstruction2 VDD VSS VDD VSS "
  ]
}
```

## A.6 Top-Level Mixed Signal Simulation Testbench

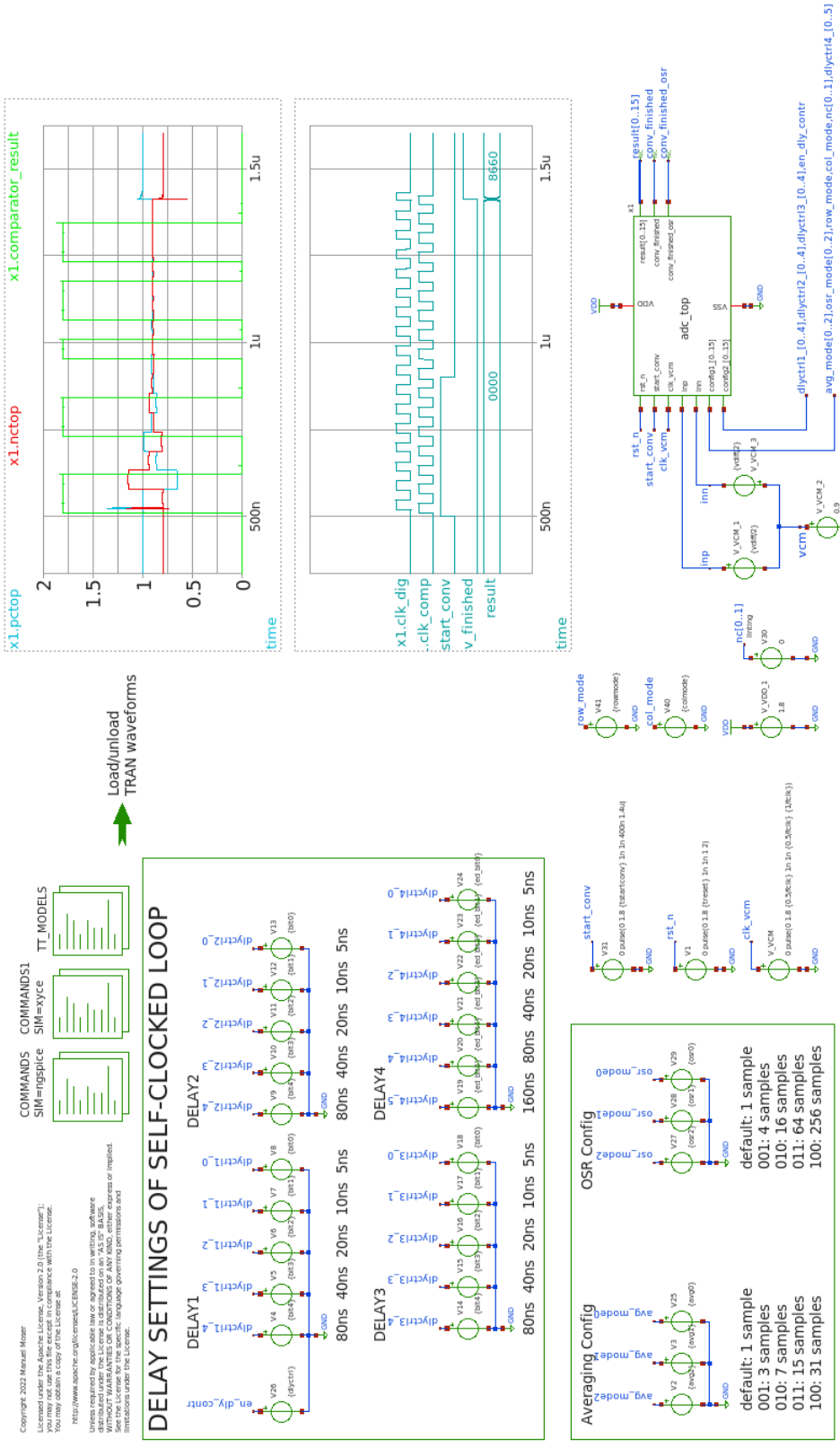



**Figure A.1:** Testbench of the top-level transient mixed-signal simulation.

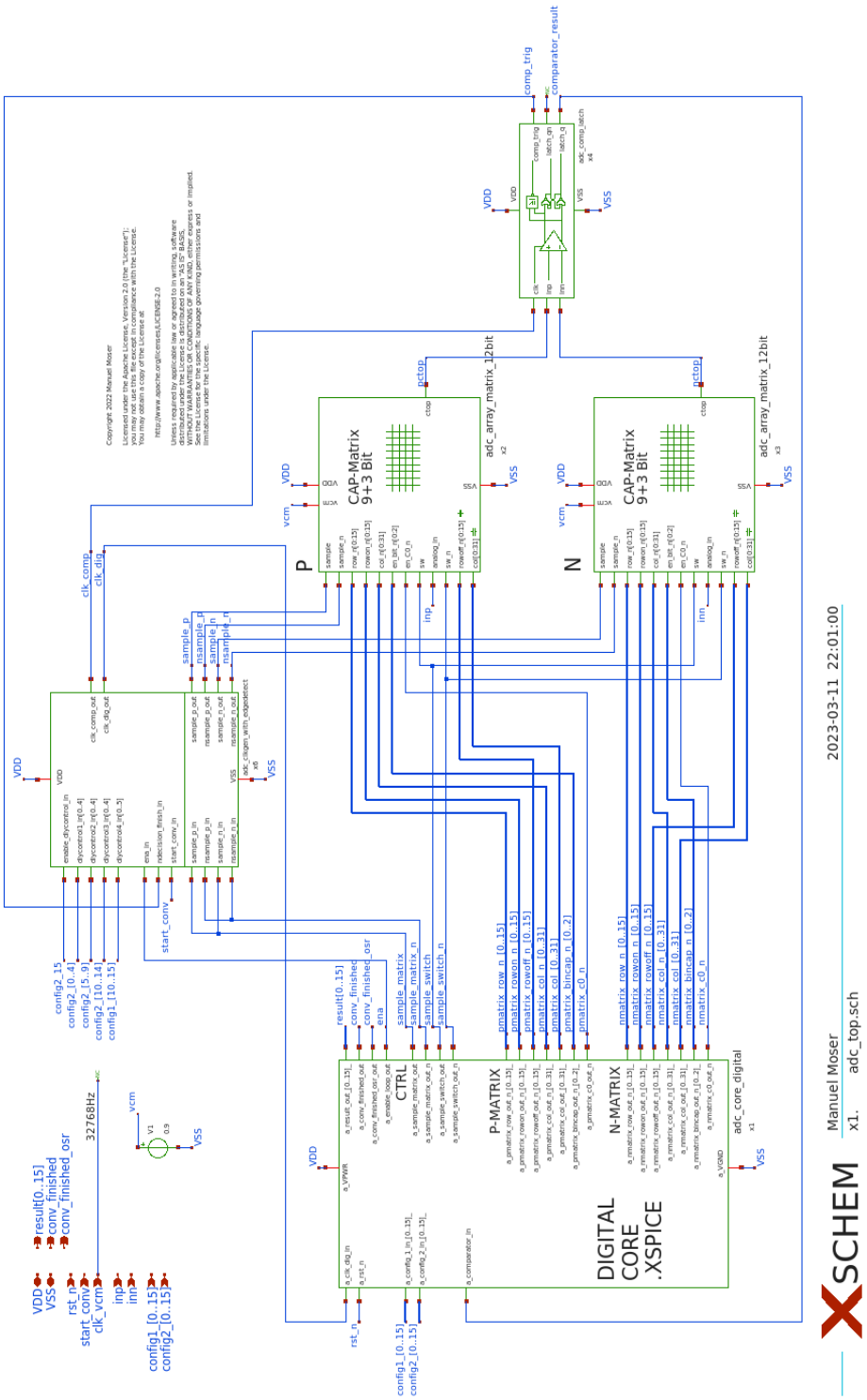



**Figure A.2:** Subcircuit of the `adc_top` symbol in the `adc_top` transient top-level testbench.

# References


[1] Efabless Corporation. *MPW-8 Summary*. 2023. URL: https://platform.efabless.com/shuttles/MPW-8 (visited on Jan. 9, 2023).

[2] Efabless Corporation. *MPW-1 Summary*. 2023. URL: https://platform.efabless.com/shuttles/MPW-1 (visited on Jan. 9, 2023).

[3] H. Pretl. *Efabless Submission SAR-ADC and Digital Temperature Sensor*. 2023. URL: https://platform.efabless.com/projects/1538 (visited on Jan. 9, 2023).

[4] *Skywater-PDK Readme*. 2023. URL: https://github.com/google/skywater-pdk/blob/38aa82e0e8563628eb3630899592ca921c70803c/README.rst (visited on Jan. 11, 2023).

[5] *Introduction to the SkyWater PDK*. 2021. URL: https://isn.ucsd.edu/courses/beng207/lectures/Tim_Edwards_2021_slides.pdf (visited on Jan. 31, 2023).

[6] *Comment by Tim Edwards in open-source silicon.dev Slack on MOSFET-size*. 2021. URL: https://open-source-silicon.slack.com/archives/C016UL7AQ73/p1645192790078449?thread_ts=1645136444.466349&cid=C016UL7AQ73 (visited on Jan. 31, 2023).

[7] *Skywater-PDK Digital Libraries*. 2023. URL: https://github.com/google/skywater-pdk/blob/a1492be33ebd1c7474f66ff76f8bdba6b01c2702/docs/contents/libraries.rst (visited on Jan. 11, 2023).

[8] T. Kugelstadt. "The operation of the SAR-ADC based on charge redistribution". In: *Analog Applications Journal* (Feb. 2000), pp. 10–11.

[9] S. Schmickl, T. Faseth, and H. Pretl. "An Untrimmed 14-bit Non-Binary SAR-ADC Using 0.37 fF-Capacitors in 180 nm for 1.1 $\mu$W at 4 kS/s". In: *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2020.

[10] F. Kuttner. "A 1.2V 10b 20MSample/s non-binary successive approximation ADC in 0.13/spl mu/m CMOS". In: *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No.02CH37315)*. Vol. 1. 2002, pp. 176–177.

[11] S. Schmickl. "Design Methodology and Development of Battery-Less SoCs for Bio-Sensing". PhD thesis. Institute for Integrated Circuits, Johannes Kepler University, Linz, Apr. 2021.

[12] Tomohiko Ogawa et al. "Non-binary SAR ADC with digital error correction for low power applications". In: *2010 IEEE Asia Pacific Conference on Circuits and Systems*. 2010, pp. 196–199.

[13] H. Pretl. *GitHub IIC-OSIC*. Institute for Integrated Circuits, Johannes Kepler University Linz. 2023. URL: https://github.com/iic-jku/iic-osic (visited on Jan. 31, 2023).

[14] *Skywater-PDK Device Details*. 2023. URL: https://skywater-pdk.readthedocs.io/en/main/rules/device-details.html (visited on Jan. 23, 2023).

[15] B. Razavi. *Design of Analog CMOS Integrated Circuits*. McGraw-Hill Education, 2016.

[16] B. Razavi. "The Bootstrapped Switch [A Circuit for All Seasons]". In: *IEEE Solid-State Circuits Magazine* 7.3 (2015), pp. 12–15.

[17] Michiel van Elzakker et al. "A 1.9$\mu$W 4.4fJ/Conversion-step 10b 1MS/s Charge-Redistribution ADC". In: *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. 2008, pp. 244–610.

[18] *Adjusting and Calibrating Out Offset and Gain Error in a Precision DAC*. 2023. URL: https://www.analog.com/en/technical-articles/adjust-and-calibrate-offsetgain-error-in-precision-dac-calculation--maxim-integrated.html (visited on Feb. 3, 2023).

[19] Pedro Toledo et al. "Re-Thinking Analog Integrated Circuits in Digital Terms: A New Design Concept for the IoT Era". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 68.3 (2021), pp. 816–822.

[20] *Skywater-PDK Standard Cell Libraries*. 2023. URL: https://skywater-pdk.readthedocs.io/en/main/contents/libraries/foundry-provided.html (visited on May 8, 2023).

[21] *OpenLane GitHub*. 2023. URL: https://github.com/The-OpenROAD-Project/OpenLane (visited on Feb. 7, 2023).

[22] *Comment by Tim Edwards in open-source silicon.dev Slack about XSPICE timing parameters*. 2023. URL: https://open-source-silicon.slack.com/archives/C016UL7AQ73/p1661871228902469?thread_ts=1661869406.769339&cid=C016UL7AQ73 (visited on Mar. 6, 2023).

[23] T. Miki et al. "An 80-MHz 8-bit CMOS D/A converter". In: *IEEE Journal of Solid-State Circuits* 21.6 (1986), pp. 983–988.

[24] *OpenRoad documentation*. 2023. URL: https://openroad.readthedocs.io/_/downloads/en/latest/pdf/ (visited on Mar. 14, 2023).

[25] *SkyWater Receives Funding from DOD, Partners with Google to Facilitate Open Source Design for its new 90 nm Technology Offering*. 2023. URL: `https://www.skywatertechnology.com/skywater-receives-funding-from-dod-partners-with-google-to-facilitate-open-source-design-for-its-new-90-nm-technology-offering/` (visited on Mar. 29, 2023).

[26] Micah Tseng Dr. Aubrey Beal Dr. Phillip Bailey. *10b SAR ADC developed at the University of Alabama*. 2023. URL: `https://github.com/UAH-IC-Design-Team/sky130-10-bit-SAR-ADC` (visited on Mar. 28, 2023).

[27] Y. Zha et al. "An Untrimmed PVT-Robust 12-bit 1-MS/s SAR ADC IP in 55nm Deeply Depleted Channel CMOS Process". In: *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2019, pp. 13–16.

[28] Min-Jae Seo et al. "A Single-Supply Buffer-Embedding SAR ADC with Skip-Reset having Inherent Chopping Capability". In: *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2019, pp. 189–192.

[29] Jae Sik Yoon, Jiyoon Hong, and Jintae Kim. "A Digitally-Calibrated 70.98dB-SNDR 625kHz-Bandwidth Temperature-Tolerant 2nd-order Noise-Shaping SAR ADC in 65nm CMOS". In: *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2019, pp. 195–196.

[30] Hugo França et al. "A 100nW 10-bit 400S/s SAR ADC for ultra low-power bio-sensing applications". In: *2017 6th International Conference on Informatics, Electronics and Vision & 2017 7th International Symposium in Computational Medical and Health Technology (ICIEV-ISCMHT)*. 2017.